GUIgen

GUI generation in the Forth style

Graham Smith, Tectime Data Systems

Published by Microprocessor Engineering

GUIgen
User manual
Manual revision 3.1
27 October 2014

Software
Software version 3.1

For technical support
Please contact your supplier

For further information
MicroProcessor Engineering Limited
133 Hill Lane
Southampton SO15 5AF
UK

Tel: +44 (0)23 8063 1441
Fax: +44 (0)23 8033 9691
e-mail: mpe@mpeforth.com
tech-support@mpeforth.com
web: www.mpeforth.com

Table of Contents

1 Introduction to Windows windows using VFX Forth

1.1 Rationale

When I first started to use MPE's VFX Forth for Windows (or more simply 'VFX'), I soon found that to create a window in Windows was much more difficult than creating a window in (MS)DOS. What used to be a one-liner in MPE's Modular Forth became perhaps a dozen or more in VFX. VFX uses its inbuilt BNF parser as a "scripting tool" to describe and define windows. I wondered at this approach which seemed to me a very non-Forth one.

I determined to create my own window wordset - one that would be both "simple" (back to the one-liners) and "open" (allow anyone to change it) and extensible. Neither must it be difficult to use, and to this end I aimed at a final syntax similar to MPE's (which was in turn similar to Windows format resource scripts).

These are the results of my labours and what is quickly apparent is that I did not manage "simple". The code is open however - if you change it please let me or MPE know. Deviations from the Windows resource script format are being introduced as new capabilities are being added.

A number of features have emerged which to me seem generally and genuinely useful. These are (in no particular order):

- Window classes, windows, dialog boxes and controls are each defined in a similar way.
- The difference between "modal" and "modeless" dialog boxes is largely hidden.
- All Windows controls are "automatically" subclassed which allows you to add control-specific message handlers simply without dealing with subclassing in Windows.
- New window controls can be defined extremely easily.
- Controls can be "placed within" dialog boxes as is usual but in addition they can be placed within normal windows and within other controls.
- Each window, dialog box and control has its own private data area.
- Words which are designed to be executed inside a WndProc procedure can be tested at the keyboard.

The rest of this document is a brief introduction to using GUIgen for Windows programming.

1.2 Windows Programming

The Windows method.

From the programmer's perspective, Windows can be thought of as a collection of libraries which provide a miriad procedures each one performing some small task. Many of them must be used to display a window on the screen, and many more learned in order to do anything more useful. These procedures make up the Windows application programming interface (API) and the phrase 'API call' is often used to indicate execution of one of thes procedures.

Of course, one of the reasons for using GUIgen is to avoid using API calls, so they are mentioned little in what follows, usually when forming part of an explanation or where an alternative is not available.

The Windows "event-driven" paradigm must be understood, to some degree at least, before a program of any complexity is created. The idea is not difficult but it should be kept in mind, particularly if it is new to you. It is this: Windows tells YOU (your program) when something happens and if you are interested you may do something about it.

For instance, when a user places the mouse pointer over a button on the screen and then presses the mouse button your program does .... NOTHING. Your program doesn't know where the mouse pointer is or that a mouse button has been pressed. At least not yet. Instead Windows (which is continuously monitoring the mouse) detects the button click and sends you a message informing you of the event. (Actually many more messages than that are sent, but they can be ignored for simplicity).

That is how Windows works and it is worth comparing it with a more traditional approach. Typically a program would wait for a key press and when it arrives do some appropriate action. In other words, you decide when something has happended - your are not told.

The implications of this are enormous: application programs spend most of their time idly waiting to be told that something has happened (e.g. a mouse click). Someone described it as "responding to events". In some ways it is a bit like responding to a processor interrupt (do nothing much until the interrupt signal is "fired"). Perhaps the ideas are not too foreign to Forth programmers after all.

Windows messages.

Windows sends messages to individual windows by calling its so-called *window procedure* or *WndProc*. Called a *callback* function, a window procedure is part of an application but is called by Windows (thus "calling back"). Its purpose is to perform some action dependant on the message number sent to it (one of the 4 input values) and return a single result.

The messages sent in this way to a window procedure are denoted by symbolic constants often with the prefix "WM_" (a contraction of Windows Message). For example the value WM_KEYDOWN is sent when a keyboard key has been pressed. You will need to obtain a list of these messages and what they mean.

The three other values supplied with each message are the handle of the window to which the message is directed and the two values referred to as *wParam* and *lParam*. The 'l' indicates a long (32-bit) value, 'w' indicates a word (16-bit) value. The names are inherited from older versions of Windows. On 32-bit Windows both parameters are 32 bits. The information held in the values depends on the message (number) sent.

Message Chains

A window procedure most usually takes the form of a *case* statement - in many cases a very long one. GUIgen uses an Similar to MPE's *Switch Chain* mechanism, these *Message Chains* provide a simple and convenient way to associate forth words to window messages and provide an effective method of managing inheritance of message response.

The following code is an example of assigning a particular action to an event. In this case the word `mydlg-HasFocus` will be executed when a `WM_SETFOCUS` message is received.

```
    WM_SETFOCUS does mydlg-HasFocus
```

This next example shows the use of `inherited` which performs a previously assigned action. The WM_GETDLGCODE message is 'intercepted' and DLGC_WANTCHARS 'added' to whatever value was returned.

```
: myctrl-GetDlgCode \ --
    inherited       \ do the default action
    DLGC_WANTCHAR   \ the bit flag required
    ReturnVal or!   \ added to the return value
;
```

See the section *Types of Window* below and the chapters about Window, Dialog boxes and Controls for more details.

1.3 Types of Window.

Standard Windows

Each window belongs to a *window class* which specifies some "base properties" such as the background colour of the window. A window class is created with the word `WinClass:`.

Any number of windows can be created based on a single window class. In Windows a window is called an *instance* of a window class. GUIgen provides the word `Window:` to create a dictionary word and that word can be then be used to create any number of window *instances*.

GUIgen window classes each have two message chains associated with them - a class chain and a default chain. The default chain is common to all window classes and defines some default actions for a few messages. The class chain, initially empty, is populated by an application as part of the `WinClass:` definition. A `Window:`-defined word also has its own message chain, which is also entirely populated by an application. Each message chain can contain actions for the same message.

When a window receives a message, each message chain is searched for an entry corresponding to that message. The search is performed in strict order : first the window chain, followed by the class chain and finaly the default chain. Some control over the order of message chain traversal is available with `inherited` and related words.

Careful use of message chains allows two or more similar windows to be defined which share a common window class but which show slightly different behaviours. See the chapter *Creating Windows* for more information.

Dialog Boxes

The familiar dialog box is a window based on a dialog box class which is buried somewhere in Windows. The behaviour of a dialog box is thus specified by the OS which implies that dialog boxes "look and feel" similar whichever application creates them.

Windows handles a dialog box as either *Modal* or *Modeless*. In both cases when a dialog

box is created, it is made the active window (accepts user input etc.) When a modal dialog box is created, Windows enters a message loop until the dialog box is closed. That means the application "looses control" during this time. In contrast, when a modeless dialog box is created control returns immediately to the application. Modal and modeless dialog boxes are created and destroyed using different API calls, but GUIgen handles these details for you.

Dialog box behaviour can be modified by assigning message handlers to carry out different or additional actions. They are based on message chains of which there are two - the dialog chain which is searched first, and the default (dialog) chain which is searched afterwards.

Guigen allows dialog boxes to be created based on application-defined window classes which have their own private and default message chains. For more information see the chapter entitled *Dialog Boxes*.

Controls

Windows provides a number of windows classes for creating windows with specialised behaviour. For use mainly with dialog boxes these *controls* provide the familiar user input and display capabilities without applications having to invent them. Controls include *Edit boxes*, *text label*, *Combo boxes*, *list boxes* etc.

GUIgen allows these controls to be created using the word `WinControl:`. (See the chapter "Controls" for more details). Many are pre-defined within the code (e.g. edit boxes and text lables), others are supplied in seperate files along with a number of words to help manipulate them (e.g. List boxes and Combo boxes). Variations on existing controls can be acheived by using the word `Another` which might specify different window styles and/or additional event responses.

You can create your own unique controls referred to as *application controls* using the word `AppControl:`. These are standard windows based on a window class but treated just like a control supplied by Windows. A number of application controls are supplied as source code, including the *Column List* and the *Grid Control*.

Controls can be assigned their own message handlers just like standard windows and dialog boxes, thus allowing customised behaviour to be added to existing controls.

1.4 Window Properties - The WinProps structure

Using GUIgen, each window, dialog box and control created, has its own private data space known as its *window properties* or just *WinProps*. This memory is `allocate`d when the window (or dialog box or control) is created, and `free`d when the window (or dialog box or control) is destroyed. By default the size of this memory is `/WinProps` which is a MPE-style data structure containing a number of data used mainly within the depths of GUIgen. (You don't care what they are - I cannot remember anymore now GUIgen is stable).

To avoid the use of global variables you will want to use WinProps. The idea is to define a data structure which is more than `/WinProps` bytes in length and use that extended area. A simple way to do so is to use the word `/WinProps:` (See *Creating Windows* for details). The structure is 'added' to the `Winclass`, `Window: DialogBox:` or `WinControl:` word definition scripts using the word `PropertySize`.

A more recent and simpler method of extending a window's `WinProps` size is to use the word `/Prop:`. Used inside a window (or dialog box or control) definition this word takes a number from the stack indicating a size in bytes and creates a word which when executed, returns the address of an area in the WinProps of the specified size. For example, to create a cell-sized WinProps 'field' the following does the job:

```
cell Prop: MyInt
```

1.5 Interactive testing of Window Procedure.

A window procedure "takes place" in what looks to VFX like a seperate task. One of the first things that GUIgen does when it processes a window message is to assign the message parameters to Forth user variables which are 'visible' by that task only. GUIgen window procedure message handlers access these variables implicitly.

Each Forth task has its own data stack, and "stack traffic" is not easily seen from the VFX console window. A window message can be synthesised by setting the user variable in the console and then executing the word under investigation. The word `Set-CurrWin` which takes a window handle from the stack and sets the appropriate user variables is usually sufficient. If particular values returned by `wParam` and `lParam` are needed, these also can be set. (See the user variables in *Creating Windows*).

1.6 A Note on Notation

An attempt was made to repoduce VFX's resource script format where possible to allow GUIgen code to be understood by anyone familiar with VFX or with other similar resource scripts. How this was achieved is as follows.

Firstly, words used to define a resource (i.e. a window class, a window, a dialog box or a control) are compiled into a seperate vocabulary.

Secondly, the resource defining word (for example `WinClass:`), adds that vocabulary to the search order.

Words are executed to define the resource, and the search order restored when the definition is complete.

As an example, to create a window class the defining word `WinClass:` is used which after creating a dictionary entry, adds the vocabulary `DefiningWinClass` to the search order. From that moment on all the words used to define a window class are revealed, the required words are used and the search order restored by `End-WinClass`.

A similar approach was made when defining windows, dialog boxes and controls although the vocabulary in each case is a different one.

While this technique can easily be abused (All other Forth words are also available) it has the merit of being both simple and easily extensible.

2 Creating Windows

2.1 Introduction

There are two parts to creating a window: creating a window class, and creating a window - an 'instance' of that class.

The word `WinClass:` defines a window class, and `Window:` defines a window. `WinClass:` adds the vocabulary `DefiningWinClass` to the search order and `Window:` adds `DefiningWindow`. These vocabularies provide words used to define the properties of that object. This results in the use of a notation resembling a 'resource script' which may be familiar to some.

Words for manipulating dialog boxes and window controls (in DlgBoxes.fth and WinCtls.fth) build on this code. Of particular note is the ability to specify controls within both window class definitions and window definitions. In addition a window class as defined by `WinClass:` may itself be used to create a new control.

2.2 Window Procedures

Windows communicates with an application primarily through window procedures or WndProcs, typically each window class having its own. A WndProc is an example of a so-called callback function, which is a function in an application that Windows executes - allowing Windows to 'talk to' the application.

A WndProc requires 4 stack items passed to it : a handle to the window at which the communication is being made, a message number indicating the type of communication being made and two values known as wParam and lParam, which convey various data to the WndProc, dependant on the message. A single value is returned by the WndProc, its meaning differs depending on the message.

In this code a single WndProc callback function is defined and specified as the window procedure of each window class created. Its action is to store the four input values in `USER` variables before executing an application-supplied message handling word. The value returned to windows is taken from another `USER` variable.

The application-supplied message handling words mentioned above are organised as singly-linked list termed a 'message chain'. Each window class has its own message chain, which gives each window based on it the same characteristic appearance and behaviour. In addition, each `Window:` defined window has its own message chain which can add to and/or modify the class behaviour. See the section on message chains later. Message chains are also used with dialog boxes and controls.

2.3 Vocabularies

Three vocabularies are used to hold the different wordsets: `DefiningWinClass` and `DefiningWindow` contain the words used in application code to configure a window class and a window 'instance'. The `Windows` vocabulary contains the general non-public words of this code. The 'high level' words, those expected to be used in application code are compiled in the GUIGEN vocabulary using the VFX Export mechanism. (The GUIGEN vocabulary is that vocabulary which is current when GUIGEN is compiled).

`vocabulary Windows`
The vocabulary which contains the general and low-level words for this package.

`vocabulary DefiningWinClass`
Contains the words used when defining a window class. This vocabulary is placed in the search order by `WinClass:` and removed by `End-WinClass`.

`vocabulary DefiningWindow`
Contains the words used when defining a window . This vocabulary is placed in the search order by `Window:(` and removed by `End-Window`.

2.4 Window Control positioning

When defining controls, their position in the window or dialog box is relative to an origin within the client area of the window or dialog box being defined. An application sets this origin using the word `ControlOrigin`

`: ControlOrigin  \ x y --`
Sets the origin relative to which a window's controls are positioned.
See the documentation for Window Controls for examples of how use this feature.

2.5 WINPROPS - Extensible window property data structure

Managing windows often requires information about each window in addition to that held by Windows. In order to provide this, a block of memory is allocated from the heap and 'attached' to each window. The memory is accessed as a data structure defined using `STRUCT`, and is referred to as a window's WinProps. Each window, dialog box and control has at a minimum, a WinProps data structure as follows:

```
struct /WinProps
    cell field winprop.Size         \ size of properties field
    cell field winprop.WinStruct    \ address of WinStruct data
    cell field winprop.hWnd         \ Windows handle for the window
    cell field winprop.WinChain     \ address of window message chain
    cell field winprop.CtlChain     \ address of message chain list
    0    field winprop.Misc         \ Miscellaneous data
    cell field winprop.MiscHi       \   high cell of Miscellaneous data
    cell field winprop.MiscLo       \   low cell of Miscellaneous data
    cell field winprop.Datum        \ Another Miscellaneous data field
    cell field winprop.GGStyle      \ GUIGEN style bits
    cell field winprop.hPopup       \ derived from Winstruct
    cell field winprop.PopupOff?    \ true if hPopup not used
    cell field winprop.hFont        \ handle of current font set by WM_SETFONT
    cell field dlgprop.ExitCode     \ value returned from modal box
    cell field dlgprop.Modeless     \ TRUE is this a modeless box
    cell field dlgprop.hToolTips    \ handle of tooltips window
    0    field winprop.PS           \ synonym on next field
    PAINTSTRUCT
        field winprop.PaintStruct   \ Address of a Windows PAINTSTRUCT structure
    /WinState
        field winprop.WinState      \ Window size, position and 'state'
    1   field   winprop.OwnFont?    \ true if using own local font
    /Font field winprop.font        \ the font definitions for this widow
    1   field   winprop.OwnBrush?   \ true if using own local brush
```

```
    /Brush field winprop.Brush      \ brush defined for this window
    aligned

end-struct
```

- `WinProp.Size` holds the size of the structure and is not used other than as a debug/verify aid. The value increases for windows which have extra fields defined.
- `WinProp.hWnd` holds the Windows handle of the window this data 'belongs' to.
- `WinProp.WinChain` hold the address of the window message chain. Also used to store the message chain of a dialog box.
- `WinProp.CtlChain` hold the address of the control message chain.
- `winprop.Misc` is a 64-bit value which can be used by an application as required. `winprop.MiscHi` and `winprop.MiscLo` can be used to access the high and low 32-bit cells.
- `winprop.Datum` is another 32-bit data value indended for ad-hoc use.
- `winprop.GGStyle` 32 bits for use as extra 'style' bits for home-grown controls.
- `winprop.hPopup` holds the Windows handle of any popup menu associated with the window.
- `winprop.PopupOff?` is TRUE if the popup menu is not to be shown.
- `winprop.hFont` is the handle of the current font. This is set when a window or control receives a WM_SETFONT message. It is also set to the handle of any font specified when a window, or control is specified.
- `DlgProp.ExitCode` is used to contain the code returned to an application when a modal dialog box is closed.
- `DlgProp.Modeless` is a flag which is true if the dialog box is modeless.
- `DlgProp.hToolTips` is a Windows Tooltip window. A Tooltip window is created and assigned to each dialog box when it is created.
- `WinProp.PS` is a synonym for `WinProp.PaintStruct`.
- `WinProp.PaintStruct` is a paint structure which is used (by Windows mostly) when a WM_PAINT event is being handled. Applications often need no knowlege of this structure. It is included here because a WM_PAINT message handler is REQUIRED to provide one and providing it here makes it unnecessary to allocate/free one at run-time. This structure is used by `Start-Painting` and `Finish-Painting`. (see `Start-Painting` later.)
- `winprop.WinState` is a `WinState` structure which is use to record the Window size, position and 'state' of the window. This structure can be saved by an application and used to restore a window's state at a later date. See `GetWinState` and `SetWinState` later.
- `winprop.OwnFont?` is true if the window or control creates its own font to use when printing. This is accomplished by specifying a font when the window (or dialog box or control) is defined using the `[Font ... Font]` notation.
- `winprop.font` holds the specification of the font to be used with this window.
- `winprop.OwnBrush?` is true if a brush has been defined for a window, doalog or control using the `[Brush ... Brush]` notation.
- `winprop.Brush` holds the definition of the brush defined for this window.

This basic `/WinProps` structure can be extended for any particular window class, window instance, dialog box or control. For instance, the following code will do:

```
struct /MyWinProps
    /WinProps   field   >baseprops  \ space for the base properties
    cell        field   MyProps.Int \ an extra integer
    32          field   MyProps.Str \ an extra string
    ... etc
end-struct
```

Winprop data fields defined in this way are usually accessed while handling a Windows message using the word `cwp` (a contraction of Current WinProps) which returns the base address of the winprops structure. For example

```
123 cwp MyProps.Int !
```

stores 123 into the MyProps.Int field. See the section about user variables below for more information about `cwp`.

The new property size must be specified as part of the windoc class, window, dialog or control in which it is required. To do this use the word `PropertySize` while defining that object.

```
WinClass: MyWindowClass
   /MyWinProps PropertySize
   ...
```

`: WinProps:`

Begin an extended WinProps structure definition. This word will be removed some time in the future.

`: WinProps  \ hWnd -- addr`

Returns the WinProps data structure of the given window. That is, it retrieves the window property with name string `"GUW-WinProps"`.
As from December 2011, the word `PROP:` should be used while defining a window class, a window, a dialog box or a control to define extra WinProps data fields. `PROP:` is a defining word for creating data fields appended to the winprops of the item being defined. It takes a single number from the stack indicating the size of that field. When executed it returns the address of the winprops field of the *current window* without the use of `cwp`.

The benefits of using `PROP:` are two-fold: firstly, it is simpler than creating a seperate structure and then specifiying its size. Secondly, `PROP:` appends fields to the item currently being defined; there is no requirement to know what the default winprops size is.

As an example the previous winprops fields defined above can be defined using `PROP:` in the following way.

```
WinClass: MyWindowClass
   cell    Prop: MyProps.Int
   32      Prop: MyProps.Str
   ...
```

In this case the data is accessed (while servicing a window message) as follows:

```
123 MyProps.Int !
```

```
: Prop:     \ n "name" ++ (compiling); -- addr (child)
```

Create a WinProp field for the window currently being defined. The field is n bytes long and its address is returned when the word is executed during a window message handler.

2.6 WinParams - Window procedures and Forth User variables

A window procedure appears to Forth as a seperate task, with its own stack and user area. This fact makes it possible to store the parameters to and from a window procedure in the User area as user variables. Doing this removes the four input parameters from the stack, and so avoids a certain amount of 'stackrobatics'.

With a few exceptions, application code will have no need to modify these values, it will simply require their stored values. For this reason a set of words is provided to do this and is documented below.

An application may sometimes need to specify the value which is returned (to Windows) by the window procedure. This value is also stored in a user variable the address of which is given by `ReturnVal`.

A window procedure takes 4 input parameters, and returns a 32-bit result. This code encapsulates these values into a single structure as follows:

```
struct /WinMsg
    cell    field winmsg.hWnd       \ the handle of window
    cell    field winmsg.msg        \ the message number
    union
      cell  field winmsg.wParam     \ the wParam value
    part
      _word field winmsg.wParamLo
      _word field winmsg.wParamHi
    end-union
    union
      cell  field winmsg.lParam     \ the lParam value
    part
      _word field winmsg.lParamLo
      _word field winmsg.lParamHi
    end-union
    cell    field winmsg.Return     \ the return value
end-struct
```

```
/WinMsg +User <WinMsg>
```

The user area which holds the current window procedure message parameters.

```
2 cells +User  <CurrWin>
```

A User Variable which holds the current window handle and the address of its window properties. The current window is normally that window which is the 'target' of a window procedure. i.e. the current window handle is the same value as is held in the winmsg.hWnd field of the current window message.

The current window can be set independently of any window message by executing `Set-CurrWin \ hWnd --` .

`cell +User <CurrDC>`
A User Variable which is used to hold a handle of a device context. It is set by `Start-Painting` and `Open-DC`, as the device context of the current window, but otherwise may be set by the application to be a printer device context or any other device context.

`cell +User NextMsgChn`
Holds the address of the next message chain to execute.

`2 cells +User <WinMsgHandler>`
Holds the xt of a default windows message handler for the window procedure (including dialog/control procedures) being executed. The second cell holds a copy of this xt which allows the primary cell to be changed and restored with the words `-WinHandler` and `+WinHandler`.

Accessing the user variables.

The words above will rarely be used by aplication code. The following words provide conveient access to the user variables for use in application code.

`: CWP \ -- addr 016 May-05`
Returns the WinProps of the current window - that is the window whose handle is held in the user variable <CurrWin>.

`: CurrWin \ -- hWnd`
Returns the value held in the user variable `<CurrWin>` which is set during a window procedure. The value can be set by an application using `Set-CurrWin`.

`: Set-CurrWin \ hWnd --`
Sets the current window/winprops user variable.

`: ?CurrWin \ -- hWnd | 0`
Returns the current window handle or 0 if it is not a valid window.

`: lParam ( -- u)`
Returns the lParam value for the window procedure being executed.

`: lParamLo ( -- w)`
Return the Lo word part of the current lParam value.

`: lParamHi ( -- w)`
Return the Hi word part of the current lParam value.

`: wParam ( -- u)`
Returns the wParam value for the window procedure being executed.

`: wParamLo ( -- w)`
Return the Lo word part of the current wParam value.

`: wParamHi ( -- w)`
Return the Hi word part of the current wParam value.

`: ReturnVal \ -- addr`
The address at which holds the value to be returned to windows on termination of the current window procedure.

`: CurrDC \ -- hDC`

Returns the handle to the device context held in the user variable <CurrDC>. This value is set by the words `Start-Painting` and `Open-DC` and an application can set it using `Set-CurrDC`.

`: Set-CurrDC  \ hDC --`
Sets the Current device context user variable. `Init-DC` can then be used if the device context 'belongs' to the current window.

`: SetWinParams  \ hWnd msg wParam lParam --`
Places the values *hWnd msg wParam* and *lParam* in their relavent User Variables and calls `Set-CurrWin`. A factor of `Init-WndProc`.

`: GetWinParams  \ -- hWnd msg wParam lParam`
Retrieves the WndProc parameters from the User Variables they were stored in by `Save-WinParams`. The values are placed on the stack in the order supplied to a window procedure, so this is the word to use before calling a default internal Windows procedure.

`: Init-WndProc  \ hWnd msg wParam lParam -- ;`
The first action performed by the window procedure, dialog procedure and control procedure. The four parameters are stored in their respective user variables and `ReturnVal` is set to zero. The message chain is initialised to a *no operation* and will require further setting up depending on the particular procedure being executed.

`: RelayMessage  \ hWnd -- u ;`
Send the current message to the given window by calling the Windows API function SendMessage.
The WinState structure is defined as follows:

```
struct /WinState
    cell    field   winstate.left
    cell    field   winstate.top
    cell    field   winstate.width
    cell    field   winstate.height
    cell    field   winstate.state  \ 0 = normal, 1 = minimised, 2=maximised
end-struct
```

`: GetWinState  \ hWnd addr --`
Copy the current WinState values to the buffer given at address *addr*.

`: SetWinState  \ addr hWnd --`
addr is the address of a `/WinState` structure holding information which is used to set the state of the given window.

2.7 Creating new messages.

A window procedure is called in response to something happening or in order to obtain information. The message number supplied to a window procedure specifies the event or required action and also specifies what the return value is to indicate.

Messages in the range 0..WM_USER-1 are reserved by Windows. The common Windows messages such as WM_CREATE and WM_COMMAND are in this range.

Windows allows the creation of private messages which range in value from WM_USER up to $7FFF. These values can be used within a private window class. Some predefined window classes already use values in this range (e.g. Buttons, edit boxes and others). WM_USER has the value $400.

A third range from WM_APP ($8000) to $BFFF is available for use by applications and do not conflict with system messages. To avoid an application needing to remember which numbers in this range remain available an application should use the word AppMessage: described below.

Two other ranges are defined: $C000-$FFFF are defined at run time when an application calls RegisterWindowMessage. Message numbers greater than $FFFF are reserved by Windows.

```
: AppMessage:   \ "<name>" ++ (compiling); -- n (child);
```
A word to create a new system message. This word creates a constant greater than WM_APP which can be used as an application defined message. Succesive use of this word produces constants each one of which is greater then the previous by one.
GUIGen defines so-called Control Notification messages: these are certain Windows messages which are sent from a parent window to the child window to which the message refers. To distinguish these messages from normal window messages, the value CN_MSG0 ($B000) is added to the Windows message number.

A number of control notification messages are defined, CN_NOTIFY and CN_COMMAND which are sent in response to WM_NOTIFY and WM_COMMAND messages respectively. For owner-drawn controls CN_DRAWITEM and CN_MEASUREITEM messages are sent in response to the WM_DRAWITEM and WM_MEASUREITEM messages.

Applications can control the colour of some Windows controls by responding to a set of messages sent by Windows to the parent window of those controls. GUIgen intercepts each of these messages and sends equivalent Control Notification messages to the control in question. This allows a degree of independance to be created for certain controls.

These Windows messages have names begining WM_CTLCOLOR and the corresponding control notification messages are named CN_CTLCOLOR.

For each WM_CTLCOLOR... message, *wParam* is the handle to a device context and *lParam* is the handle of the control. An application sets the font and background colour of the device context to suit. A valid brush hanle **must** be returned if Windows is to accept the changes to the device context. The brush is used to erase the device context.

GUIgen allows the definition of a Windows control which performs this colour control by answering the GUIgen CN_CTLCOLOR... message. That control can be used within any number of parent windows which do no now need to service the WM_CTLCOLOR... message.

```
$B000 constant CN_MSG0
```
The 'base' of the Control Notification message numbers

```
: CNMessage:        \ n "name" ++ (compiling); -- n' (child)
```
Defines a Control Notification message. *n'*, the value returned is equal to *n* plus `CN_MSG0`.
The following Control Notification messages are defined:

```
WM_COMMAND              CNMessage:  CN_Command
WM_NOTIFY               CNMessage:  CN_Notify
WM_DRAWITEM             CNMessage:  CN_DRAWITEM
WM_MEASUREITEM          CNMessage:  CN_MEASUREITEM
WM_CTLCOLOREDIT         CNMessage:  CN_CTLCOLOREDIT
WM_CTLCOLORLISTBOX      CNMessage:  CN_CTLCOLORLISTBOX
WM_CTLCOLORBTN          CNMessage:  CN_CTLCOLORBTN
```

```
WM_CTLCOLORSCROLLBAR    CNMessage:  CN_CTLCOLORSCROLLBAR
WM_CTLCOLORSTATIC       CNMessage:  CN_CTLCOLORSTATIC
```

Two associated messages are not handled in this way. WM_CTLCOLORMSGBOX, though documented is never sent and WM_CTLCOLORDLG is sent directly to a dialog box and which guigen dialog boxes handle 'themselves'.

`: NotifyControl     \ hWnd -- ;`

Sends a Control Notification message to the window with the given handle. The message number sent is equal to the current message number plus CN_MSG0. If hWnd is zero, no action is taken. Any returned result is assigned to ReturnVal.
For instance when a button is clicked, its parent is sent a WM_COMMAND message. GUIGen sends a CN_COMMAND message to the button allowing it to carry out its own action.

2.8 Message Chains

The Message Chain window message handler

A message chain is a singly-linked list of message number and *xt* pairs. A unique message chain is typically created for each window class and another for each defined window. Similarly, each dialog box and window control has its own unique message chain.

When Windows sends a message to a GUIgen window (or dialog box or control), GUIgen traverses the message chain looking for an item with a matching message number and if such a match is found the associated *xt* is executed.

In practice two or more message chains are linked together in a 'chain of chains'. For any message, each chain is searched in turn executing the *xt*'s when the message is found. GUIgen places a window message chain before its window class message chain allowing a paricular window to change or add to, its default (i.e. window class) behaviour. After all messages chains have been searched, the default window procedure **WinHandler** is executed.

As stated above, each `WinClass` contains a message chain and so does each `WinStruct`. Initially empty, items are added to a message chan using the words `Does` and `Does:` inside `Winclass:` and `Window:` definitions.

The example below shows part of a window class definition. Any window created based on this window class will perform *InitialiseMyWindow* when it receives a WM_CREATE message, and will beep 5 times whenever the left mouse button is pressed.

```
WinClass: MyWindowClass
    ... other commands defining the class ...
    WM_CREATE       does    InitialiseMyWindow
    WM_LBUTTONDOWN  does:   5 0 do 0 MessageBeep drop loop ;
end-winclass
```

Of course either of these message handling assignments could have been made within a `Window:` definition instead with similar results.

Modified Behaviour and Inheritance

When a window receives a message its `WinStruct` message chain is searched first followed by its `WinClass` chain. There is also a third chain (`DefMsgChain`, see later) which is searched last and is common to all windows to perform some default processing.

Providing multiple chains allows an application to add and/or modify a windows behaviour (its actions in response to a message), with the help of a few extra words. To explain this more fully we must look at the traversal of message chains in more detail.

As has already been stated, an action is performed (a word executed) if a sought-for message number is found during a traversal of a message chain. Now note that after a message is found and the matching action performed the search is terminated. Thus adding a message handler to a message chain effectively **replaces** any previous handler! This can be inconvenient when wishing to add to an existing action.

One solution is to create a window class which describes the basic or common behaviour and provide additional message handlers in the `Window:` definition. In this case the window handler for a particular message is executed, followed by the window class handler.

To cause a WinClass: action to be performed *before* a Window: action, the Window: message handler executes `Inherited` before carrying out its own actions. In this case, no further processing takes place when the Window: handler word finishes.

The message handler acts on a chain of message chains, searching each one in turn - firstly the window message chain, followed by the class message chain and finishing with the default chain. Finally, any default action required by Windows is performed - the Windows API function `DefWindowProc` is called. `Inherited` works by causing the chain-of-chain search to continue prematurely, thus carrying out actions which the window *inherits* from its class definition.

Occasionally it is useful to avoid any inherited behaviour and `-Inherited` does this by teminating the traversal of the chain of chains. Most importantly the default Window processing is also inhibited and if this is not desired use `+WinHandler` will reinstate the action.

Managing Message Chains

A message chain forms part of both `WinClass:` and `Window:` data structures, and each word records its own message chain as being the *current message chain*. (See `CurrMsgChain` below). The use of both `Does` and `Does:` is found within `WinClass:` and `Window:` 'definition scripts' since they act on the current message chain. But additional words are available to allow the words to be used elswhere.

The 'head' of a message chain and each item in the chain are defined as follows:

```
struct /MsgChain
    cell field msgchn.Next      \ address of next chain
    cell field msgchn.FirstItem \ address of first item in chain
end-struct
```

```
struct /MsgItem
  cell field MsgItem.next   \ address of next item in list
  cell field MsgItem.id     \ id for this item
  cell field MsgItem.xt     \ xt to run
end-struct
```

The following actions relating to message chains are defined:

`: .msgitem      \ addr --`
Print the ID and *xt* address of a message chain item.

`: .chain        \ msgchain --`
list the message chain items.

`: .Chains       \ msgchain -- ;`
Lists the chain of chains.

`: ?MsgDup   \ msg# chain --`
Searches the given message chain for a message item for the given message ID. A warning is given if a match is found. Used by `DOES` and `DOES:`

`: MsgItem,      \ id -- addr`
Create a message item at HERE returning its address. The MsgItem.xt field is set to the xt of NOOP, and the MsgItem.next field is zero.

`: InsMsgItem    \ MsgItem Chain -- ;`
Insert message item into the begining of the given message chain.

`: AppChain  \ chain1 chain2 --`
Append the message chain *chain1* to the end of the message chain list begining with *chain2*. For example if chain2 points to chain3 the before and after situations are as follows:

```
Before: Chain1 --> x    Chain2 --> Chain3 --> Chain4 --> 0
After:                  Chain2 --> Chain3 --> Chain4 --> Chain1 --> x
```

`code ExecChain  \ i*x ID chain -- j*x`
Traverse the message chain for the given ID and execute the associated execution token if found. If a matching ID is not found the effect is 'no action' (apart from removing *ID* and *chain* from the stack).

Any action performed when a matching *ID* is in response to a Windows message. It will typically inspect the `wParam` and `lParam` values, perform some operation but have no net stack effect.

`variable <MsgChain>     \ -- addr`
Holds the address of the current message chain which will be acted upon by `DOES` and `DOES:`. The current message chain is set by both `WinClass:` and `Window:`. It is also set when defining dialog boxes and controls.

The other words of this set access this word indirectly through `CurrMsgChain`.

`: CurrMsgChain     \ -- addr`
Returns the address of current message chain held in `<MsgChain>`. Does an ABORT if the value is zero.

`: does:     \ msg# "words.." ++ ; add nameless action to message chain`

Adds a new entry to the current message chain. The words between `DOES:` and `;` are compiled and this un-named word is executed when a message with ID of msg# is received. The following code defines the action for a WM_SETFOCUS message.

```
WM_SETFOCUS does: 1 #focused +! ;
```

Message chains can also be 'programmed' outside window and window class definitions. See `[Messages, Messages]` and friends below.

2.9 The WinClass structure.

/WinClass is a data structure which encapsulates the Windows structure WNDCLASSEX and is the structure created by the word `WinClass:`.

/Winclass has the following definition:

```
Struct /WinClass
    cell        field winclass.ID       \ allows us to identify this structure. ID MUST be f
    cell        field winclass.Next     \ points to next WinClass in chain
    WNDCLASSEX  field WinClass.WNDCLASSEX
    Max_Path    field WinClass.ClassName
    cell        field WinClass.ClassAtom
    cell        field WinClass.PropSize
    cell        field WinClass.FirstCtl     \ address of first control
    \ some fields of WNDCLASSEX must be filled in at run-time
    \ so keep source identifiers here...
    cell        field WinClass.IconSource
    cell        field WinClass.IconSmallSource
    cell        field WinClass.CursorSource
    /Brush      field WinClass.Brush
    cell        field WinClass.BrushDefined
    cell        field WinClass.Menu
    /MsgChain   field WinClass.MsgChain
end-struct
```

The fields have the following uses :

WinClass.ID: Allows the identification of WinClass structures and to distinguish between them and other structures.

WinClass.WNDCLASSEX: A Windows WNDCLASSEX structure which ** is used when class is registered with Windows.

WinClass.ClassName: A string which holds the name of the window class. This string can be set manually but is initialised to be a "known" unique string when the class is first defined using the word NextClassName.

WinClass.ClassAtom: Is the value which Windows returns when a window is registered. A zero value indicates that the window has not been registered.

WinClass.PropSize: Holds the size in bytes, of the WinProps data structure. (See WinProps later.)

WinClass.FirstCtl: Holds the address of the first control contained in this window class. When a window of this class is created, a number of controls will be created starting with the one at the address held here.

WinClass.IconSource: Holds the address of an Icon data structure which is built using ICON: from the Bitmaps.fth source file.

WinClass.IconSmallSource: Holds the address of an Icon data structure which is built using ICON: from the Bitmaps.fth source file. This icon is a small-sized icon.

WinClass.CursorSource: Holds the address of an icon structure to use as a cursor, built using the word ICON: (See Bitmaps.fth)

WinClass.Brush: Contains information defining the brush used to colour windows of this class.

WinClass.BrushDefined: A value of *true* indicates that the **WinClass.Brush** structure holds valid information.

WinClass.Menu: The address of a GUIgen-defined window menu which will be created for each menu created of that class.

WinClass.MsgChain: The address of the class message chain for windows of this class type.

```
: .WinClass     \ winclass --
```
For testing/debugging - prints the values of the various /WinClass fields.

```
0 value CurrWndClass
```
The window class which is being defined. Words which set/manipulate the WINCLASS.xxxx fields use this value, which is set by the word WinClass:

```
: NextClassName     \ -- add len
```
Returns a 'unique' window class name of the form `"[xxxxxxxxxxxxxxx]GUIgen-nnn"` where nnn indicates three decimal digits and [xxxxxxxxxxxxxxx] is a number generated at compile time, designed to ensure each application starts with a unique base name. Each time this word is executed, the nnn number is incremented by one.

Run-time Message Chain manipulation.

```
: -WinHandler    \ --
```
Disables the current Windows message handler. This applies to windows and controls and dialog boxes based on application-defined window classes (but not standard dialog boxes which don't have a default behaviour in the same way). Executing `WinHandler` after this is a *no operation.*

```
: +WinHandler    \ --
```
Restores the current window message handler if it has has been disabled by `-WinHandler`. Note that if this word is used within a word which is part of a message chain, there is no need to execute `WinHandler` since the message chain terminates with that action.

```
: WinHandler    \ --
```

Executes the current Windows message handler. Performs `-WinHandler` to avoid inadvertantly repeating the action. This word will usually be used within a message handler before using `-inherited` which itself makes this word a 'noop'.

`: do-DefWndProc \ --`
Call the Windows API function `DefWindowProc` using the current window procedure paramaters. The result is saved in `ReturnVal`. This is the word performed by `WinHandler`, for dialog boxes and controls `WinHandler` will do something else.

`: Inherited \ --`
Performs any inherited behaviour during a window message handling routine. Used within a Windows Procedure word, the remaining message chains within the current 'chain of chains' are traversed looking for an action to the current Windows message. (i.e. by executing `ExecChain`).

`: -Inherited \ --`
Remove any inherited actions. The chain-of-chains traversal is terminated and `-WinHandler` is executed. To remove previous GUIgen actions to a message but to perform the default Windows processing

```
WinHandler -Inherited
```

As an alternative, the following phrase has a similar effect:

```
-Inherited +WinHandler
```

2.10 Window class Registration

Windows requires that each window class be registerd before it can be used to create a window instance) . In this code, RegisterWinClass is executed 'automatically' when a window is created (by `Open-Window`), so the words described here will not normally be used by applications.

`: Ready-Icon \ WinClass --`
Initialise the hIcon field of WNDCLASSEX. An 'internal' word called by RegisterWinClass.

`: ready-SmallIcon \ WinClass -- ; 029Aug07`
Initialise the hIconsm field of WNDCLASSEX. An 'internal' word called by RegisterWinClass.

`: Ready-Cursor \ WinClass --`
Initialise the hCursor field of WNDCLASSEX. An 'internal' word called by RegisterWinClass.

`: Ready-Background \ WinClass --`
Initialise the hbrBackground field of WNDCLASSEX. An 'internal' word called by RegisterWinClass

`: Registered? \ WinClass -- flag`
Reurns TRUE if the class has already been registered.

`: RegisterWinClass \ WinClass --`
Register the given window class with Windows if it has not already been registered. Failure to register the class will show an error message on screen and return a zero in the WinClass.ClassAtom field but otherwise do nothing. NOTE: This word is executed whenever a window is created for the first time - there is no need for an application to call it under normal conditions.

`: UnregisterWinClass \ WinClass -- ior`
Unregister the given window class with Windows. Return 0 if the operation was succesfull or if the class was not already registered. A non-zero return value is a Windows error code

2.11 The WinStruct Data Structure

The /WinStruct data structure is used to store information about a particular window definition which will be referred to when the window is created. WinStruct has the following definition:

```
struct /WinStruct
    cell    field   Winstruct.ID        \ allows us to ID this structure
    cell    field   Winstruct.Prev      \ address of previously defined
    cell    Field   WinStruct.Style     \ window styles
    cell    Field   WinStruct.ExStyle   \ window exstyles
    cell    Field   WinStruct.Menu      \
    cell    Field   WinStruct.PropSize  \ size of property space
    Cell    Field   WinStruct.ClassDef  \ address of window class
    Cell    Field   WinStruct.FirstCtl  \ address of first inline control
    0       Field   WinStruct.Misc      \ to pass info to WinProps
    cell    Field   WinStruct.MiscHi    \ to pass info to WinProps
    cell    Field   WinStruct.MiscLo
    Cell    Field   WinStruct.Datum     \ pass info to WM_CREATE
    cell    field   WinStruct.GGStyle   \ 025 : GUIGEN style bits
    Cell    Field   WinStruct.Popup     \ holds address of popup
    256     field   winstruct.Captionz  \ window caption
    cell    field   winstruct.Left      \ left,
    cell    field   winstruct.Top       \   top,
    cell    field   winstruct.Width     \   width and
    cell    field   winstruct.Height    \   height of window when created.
    _byte   field   winstruct.FontDefined   \ true if a font has been defined
    _byte   field   winstruct.BrushDefined  \ true if a brush has been defined
    /Font   field   WinStruct.Font      \ the font definition for this window
    /Brush  field   WinStruct.Brush     \ the brush definition for this window

    aligned
    /MsgChain field WinStruct.WinChain  \ window message chain (and dialogs)
    /MsgChain field WinStruct.CtlChain  \ message chain for controls
end-struct


: .MsgChains        \ addr --
```

Given the address of either a

```
: MsgChain:     \ "<name>" ++
```

Creates a message chain as a named item in the dictionary. This word is used to create message chains which are set as the default for window classes. DOES and DOES: are used to add items and finish with ;MsgChain.

```
: ;MsgChain     \ -- ;
```

Terminates a `MsgChain:` definition.

```
: [messages     \ addr -- ;
```

Takes the address of a /Winclass or /Winstruct structure and sets the appropriate field as the current message chain. Using `DOES` and `DOES:` now add items to that message chain. Finish with `messages]`.

```
: Messages]     \ --
```

Ends process of appending actions to a message chain started with *\fo{[messages.

2.12 Defining Window Classes

The words `WinClass:` ... `End-WinClass` are used to create a window class. The elipses indicate words which set or alter some of the WinClass fields.

An example of a WinClass definition is as follows

```
WinClass: MyWindowClass
    CS_STANDARD Style
    IDI_ASTERISK Icon
    IDC_IBEAM Cursor
    gray_brush stock-brush
End-WinClass
```

Window Class Defining Glossary

The following words are contained in the vocabulary `DefiningWinClass` which is added to the search order by the defining word `WinClass:`.

`: Style  \ u --`
Set the current window class style to the value given. u is made up of combining one or more CS_xxx class style values. Typically use is

```
CS_HREDRAW CS_VREDRAW or Style
```

`: Style+  \ u --`
Add the given value to the current window class style using an OR operation. Typical use is

```
CS_DBLCLKS Style+
```

`: ExtraClassSpace  \ u --`
Sets the WNDCLASSEX.cbclsextra field of the current window class, to the given value.

`: ExtraWindowSpace  \ u --`
Sets the WNDCLASSEX.cbwndextra field of the current window class, to the given value.

`: Icon  \ u --`
Sets the current class icon to u. The value u will be one of the constants IDI_APPLICATION, IDI_HAND, IDI_QUESTION, IDI_EXCLAMATION, IDI_ASTERISK or IDI_WINLOGO.

23-Apr-05 : Icons created with `ICON:` (see Bitmaps.fth) can now be used in which case u is the address of a word created with `ICON:`. If `CREATE-ICON` has not been defined, the icon will not be created.

`: SmallIcon  \ u|addr -- ;  \ 029Aug07`
Sets the current class small icon to u. The value u may be one of the constants IDI_APPLICATION, IDI_HAND, IDI_QUESTION, IDI_EXCLAMATION, IDI_ASTERISK or IDI_WINLOGO, or the address of an icon structure created with `ICON:` (see Bitmaps.fth)

`: Cursor  \ u --`
Sets the current class cursor value to u which will be one of the constants IDC_ARROW, IDC_IBEAM, IDC_WAIT, IDC_CROSS, IDC_UPARROW, IDC_SIZENWSE, IDC_SIZENESW, IDC_SIZEWE, IDC_SIZENS, IDC_SIZEALL, IDC_NO, IDC_APPSTARTING or IDC_HELP. Eventually it is intended that u could be a structure defined by ICON:. (For ICON: see Bitmaps.fth)

`: [Brush \ -- brush`
Begin the definition of a class brush. A typical use might be as follows (but see documentation for Fonts and Brushes for more information).

```
[Brush
    green colour
Brush]
```

`: Brush]`
End the definition of a class brush.

`: Menu \ u --`
Sets the WINCLASS.MENU field of the current window class to the value given. This field is inspected by the default class procedure, and a menu created and associated with each instance is created.

`: PropertySize \ u --`
Sets the PropSize field of the winclass currently being defined. When an instance of this class is created u bytes of memory is ALLOCATEd, the address being stored as a Window property. This address is returned by WINPROPS.

`: End-WinClass \ --`
Marks the end of a window class definition. Resets the current window class to zero and removes the top vocabulary from the search order (using PREVIOUS).

`: | \ n1 "<name>" -- n1|n2`
The 'C' style bitwise OR operator. (The Forth word OR is available.) This word is state smart and allows the following style of coding:

```
WS_VISIBLE | WS_CLIPCHILDREN | WS_DLGFRAME Style
```

`: Prop: \ n "name" ++ (compiling); -- addr (child)`
Create a WinProp field for the window class being defined. The field is n bytes long and its address is returned when the word is executed during a window message handler.

2.13 Common Window Procedure

A generic window callback procedure is assigned to each window class. It is passed four parameters (a handle to the window, a message ID and two 32-bit values known as wParam and lParam), the values of which are assigned to the relevant User variables. This procedure also creates and initialises the WinProps structure for this window and traverses the message chains to perform the actions peculiar to the message ID. These and other common window actions form the worset in this mini glossary. Unless otherwise stated these words are 'hidden' in the Windows vocabulary and are not usually required by applications.

Window Initialisation and Finalisation

`defer Open-Attached(Win) \ WinClass --`
A word which is called by the window procedure when the window is created. Set here to do DROP, it will later be assigned to open any window controls which have been declared as being children of this window. This gives the ability to define non-dialog windows which contain controls, and it allows 'compound controls' to be built from standard windows. (See WinCtls.fth for more details).
January 2010: While GuiGen windows seem to have been working well up to now, I have discovered a problem which I do not fully understand: I created a VFX Windows DLL application

in which one exported function caused a dialog box to be created and displayed. Under most circumstances there is no problem. But when the DLL was used by a Pascal program running in the Delphi IDE closing the dialog box caused an exception referring to memory access violation. If the window property memory is not `free`d (in the word `Delete-WinProps` the exception does not occurr. The problem ONLY occurrs when using the Delphi IDE.

The following code creates a list of Window Handle/Memory Address pairs. It is used to store a list of property addresses and the window handles they belonged to. The list is added to by `ReleaseWProps` which is now used by `Delete-WinProps` instead of FREEing the property memory.

The word `FreeWProps` traverses the list and `free`s any memory whose window handle is no longer 'valid' in the expectation that it is now safe to do so.

This technique has cured the Delphi IDE exceptions. Applications can/should execute `FreeWProps` at application shut down or periodically to `Free` any memory allocations still 'lurking' around.

`: FreeWProps \ -- ;`
Frees any window property memory which is not being used by a window. End an application with this word.

`: ReleaseWProps \ props hWnd --`
Places the window properties *props* of the window with handle *hWnd* in the list of window properties which can be deleted.

`: Alloc-WinProps \ hWnd winstruct -- props`
Allocates memory for WinProps and initialises some WinProp.xxx fields. A factor of `Init-CurrWin`, it will also be used also when initialising dialog boxes and controls. *hWnd* is the handle of the window to which the WinProps memory will be assigned. *winstruct* is the address of a */WinStruct* data structure. *addr* is the address of the memory alllocated.

`: Init-CurrWin \ --`
Called in response to a WM_CREATE message to a window. The properties of the window are initialised here.

`: Delete-WinProps \ hWnd --`
Free the memory used for WinProps previously allocated by `Create-WinProps`. Called by windows, dialog boxes and controls as part of their response to a WM_DESTROY message.

`: Hdl \ ID -- hControl`
Returns the Windows handle of a control whose ID is given. The control is assumed to be a child of the 'current' window, given by CurrhWnd.

Reflecting WM_DRAWITEM and WM_MEASUREITEM messages

The parent of owner-drawn controls are sent WM_DRAWITEM messages when the control needs to be drawn. This code intercepts each WM_DRAWITEM message and sends a CN_DrawItem message to the control returning the result to Windows. A WM_DRAWITEM message handler can be defined to override this behaviour.

A similar method is employed for WM_MEASUREITEM messages, in this case CN_MEASUREITEM messages are sent to the control.

This technique will also be applied to dialog boxes and all controls - a child window can be a parent of another child window.

NOTE: Owner-draw menu items also produce WM_DRAWITEM and WM_MEASUREITEM messages. These messages do NOT generate the corresponding CN_xxx messages.

`: wproc-DrawItem \ --`
Executed in response to a WM_DRAWITEM message which is sent to the parents of controls having an *OwnerDraw* style. A CN_DrawItem is sent to the control concerned and the result returned to windows. If the WM_DRAWITEM message cencerns a menu or menu item, no action is taken.

This word is used in the same way in dialog boxes and controls.

`: wproc-MeasureItem \ --`
Executed in response to a WM_MEASUREITEM message. A CN_MEASURESELF message is sent to the originating control with the parameters wParam and lParam unchanged. No action is taken if the originating control is a menu. Also used as the default behaviour for dialog boxes and controls.

`: do-Popup \ -- ;`
The default WM_CONTEXTMENU action. Opens the popup menu associated with the current window if there is one.

A handle to the popup menu is held in windows property field *winprop.hPopup*. A zero value in this field indicates no associated menu and no action is taken. Otherwise the menu is displayed and inherited actions are stopped.

Handling Fonts

The two window messages WM_SETFONT and WM_GETFONT are recognised by Windows for controls. Here are two words which respond to these messages for standard windows to set and retrieve the winprop.hFont field of the windows properies.

`: Do-WMSetFont \ --`
The default action in response to a WM_SETFONT message. Sets the winprop.hFont to the specified font and invalidates the whole window area to cause it to be redrawn.

`: Do-WMGetFont \ --`
The WM_GETFONT message default action. Returns the value stored in the windows properties winprop.hFont field, and stops further processesing using `-inherited`.

Recording and Restoring window 'state'

The Window State is here defined as the position, width and height of a window along with information specifying whether it is minimised, maximised or not.

The field winprop.WinState records this information which changes whenever a window is resized or moved. That information can be obtained by an application and then used to restore that state at some later date. And that is the purpose of this effort - to allow a consistant way of restoring a Windows size and position by applications and to reduce the need to duplicate code for each window.

The WM_SIZE message is intercepted to record a window's width its height and its Minimised/Maximised/neither value. A window's position is recorded by intercepting the WM_MOVE message.

`: do-WM_SIZE \ -- ;`
The default WM_SIZE message handler for all windows defined by *WINDOW:*. Saves the new window state to the winprop.WinState structure.

`: do-WM_MOVE \ --`
The default WM_MOVE message handler for all windows defined by *WINDOW:*. Updates the winprop.WinState structure fields.

`: OwnFont! \ flag hWnd --`
Assigns *flag* the a window's `OwnFont?` property. When this is true a windows text is drawn using the font defined in its Winprop data.

`: OwnFont? \ hWnd -- flag`
Returns the value of a window's `OwnFont?` property.

`: WinTextColour \ hWnd -- Colour`
Returns a window's text colour as defined in its Font data in the case that `OwnFont?` is TRUE. If `OwnFont?` is FALSE the returned value is obtained from the window's parent.

`: WinTextColour! \ Colour hWnd --`
Sets a window's text colour and its OwnFont? property to TRUE.

`: OwnBrush! \ flag hWnd --`
Assigns *flag* the a window's `OwnBrush?` property. When this is true a window's background is drawn using the brush defined in its winprop data.

`: OwnBrush? \ hWnd -- flag`
Returns the value of a window's `OwnBrush?` property.

`: WinColour \ hWnd -- colour`
Returns a window's background client area colour as defined in its Brush data in the case that `OwnBrush?` is TRUE. If `OwnBrush?` is FALSE the returned value is obtained from the window's parent.

`: WinColour! \ Colour hWnd --`
Sets a window's background colour and its OwnBrush? property to TRUE.

Common default message chain

The previous words are formed into a message chain. This will be the final chain to be traversed by the window procedure after the class message chain (if any) following the window message chain (if any). It is defined as follows:

```
MsgChain: DefMsgChain
    WM_CREATE       does Create-OwnFont
    WM_DRAWITEM     does wproc-DrawItem
    WM_MEASUREITEM  does wproc-MeasureItem
    WM_SETFONT      does do-WMSetFont
    WM_GETFONT      does do-WMGetFont
    WM_CONTEXTMENU  does do-Popup
    WM_SIZE         does do-WM_SIZE
    WM_MOVE         does do-WM_MOVE
```

```
    WM_ERASEBKGND   does wproc-ERASEBKGND
    WM_MouseMove    does wproc-WMMouseMove
\    WM_CTLCOLORMSGBOX       does Send-CN-CTLCOLORxxx
    WM_CTLCOLOREDIT         does Send-CN-CTLCOLORxxx
    WM_CTLCOLORLISTBOX      does Send-CN-CTLCOLORxxx
    WM_CTLCOLORBTN          does Send-CN-CTLCOLORxxx
\    WM_CTLCOLORDLG          does Send-CN-CTLCOLORxxx
    WM_CTLCOLORSCROLLBAR    does Send-CN-CTLCOLORxxx
    WM_CTLCOLORSTATIC       does Send-CN-CTLCOLORxxx

\    CN_CTLCOLORMSGBOX       does wproc-CN-CTLCOLORxxx
    CN_CTLCOLOREDIT         does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORLISTBOX      does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORBTN          does wproc-CN-CTLCOLORxxx
\    CN_CTLCOLORDLG          does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORSCROLLBAR    does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORSTATIC       does wproc-CN-CTLCOLORxxx

    GWM-Popup       does    wproc-Popup

    GWM-SetOwnFont  does:   wparam wproc-OwnFont! -inherited ;
    GWM-GetOwnFont  does:   wproc-OwnFont? ReturnVal ! -inherited ;
    GWM-GetTextClr  does:   wproc-TextColour  ReturnVal ! -inherited ;
    GWM-SetTextClr  does:   wParam wproc-TextColour! -inherited ;

    GWM-SetOwnBrush does:   wParam wproc-OwnBrush! -inherited ;
    GWM-GetOwnBrush does:   wproc-OwnBrush? ReturnVal ! -inherited ;
    GWM-GetBrushClr does:   wproc-WinColour ReturnVal ! -inherited ;
    GWM-SetBrushClr does:   wParam wproc-WinColour! -inherited ;
    GWM-GetBrushHdl does:   wproc-WinBrush ReturnVal ! -inherited ;
;MsgChain
```

The Common Window Procedure

```
: (DefaultWinProc)  \ hdlg mesg wparam lparam -- ior
```

The action of DEFAULTWINPROC the default window callback procedure for each window class defined. Three actions are taken in the following order :

1. The four input parameters are saved as the relevant user variables and other user variables are initialised (see WndProc parameters above).
2. INIT-CURRWIN is executed if the current message is WM_CREATE.
3. Traversal of the message chain-of-chains is performed.
4. Finally, if the current message is WM_DESTROY, FIN-CURRWIN is executed to 'clean up'.

```
4 1 callback: DefaultWinProc
```

The name of the window procedure for all window classes, its action is assigned as (DefaultWinProc).

2.14 Creating a Window

Each and every window is based on a window class which must be created first. The following section describes the words used to perform these two seperate but related actions.

Defining a Window Class

`CS_HREDRAW CS_VREDRAW or CS_DBLCLKS or constant CS_Standard`
A constant used as the standard/normal style when defining a class style.

`: DefaultWinClass \ 'WinClass --`
Fill the given WndClass structure with default values. Used when vocabulary and will not be required by applications.

`: WinClass: \ "<name>" --`
The window class defining word. Takes the next blank delimited word from the input stream and creates a dictionary entry for it. Space for the `/WinClass` structure is `ALLOT`ed and the structure initialised by calling `DefaultWinClass`. Children of this word put the address of the `/WinClass` structure on the stack.

`: DefiningWClass? \ -- flag`
Returns TRUE if the current window class (the value held as the value `CurrWndClass`) is non-zero. Used in the Window Control word set.

`: UnregWinClasses \ -- 0|-1 ; ( 14Mar14)`
Unregister all defined WinClass structures. Call from a DLL when it unloads because Windows doesn't do it itself for DLLs (It does for normal programs apparently).

Setting the window properties.

Setting the fields of a window WinStruct structure is done within a `Window: ... End-Window` window definition.

`0 VALUE CurrWinStruct`
Used to hold the address of the Window/WinStruct which is currently being defined. It is set by Window: and reset by End-Window.
The following words, contained in the DEFININGWINDOW vocabulary, are used to set and modify the fields of the current window WinStruct the address of which is given by the value CURRWINSTRUCT.

`: Style \ u --`
Sets the style of the current window. u is a 32-bit set of bit flags specified by the WS_xxx window style values. An example of use is

```
WS_WISIBLE WS_POPUP or Style
```

`: Style+ \ u --`
A word to add a style to the window currently being defined. Example of use is:

```
WS_BORDER Style+
```

`: Style- \ u --`
Remove the style bit flags set in u from the window currently being defined.

`: ExStyle \ u --`
Sets the extended style of the window being defined. The value of u is defined by the WS_EX_xxx values.

```
WS_EX_WINDOWEDGE ExStyle
```

`: ExStyle+ \ u --`
Adds u to the current window exetended style. As STYLE+ but takes WS_EX_xxx values.

`: ExStyle- \ u --`
Remove the bit flags set in u from the current window's ExStyle using AND with the inverse of u.

`: PropertySize \ u --`
Sets the PropSize field of the WinStruct currently being defined. When an instance of the window is created, u bytes of memory is ALLOCATEd and its address saved. The address of that memory is returned by WINPROPS.

`: Popup \ 'popup -- ;`
Sets the current window's popup menu. 'popup is the address of a popup menu defined by POPUP:

`: Datum \ u -- ;`
Sets the Datum field of the current window to the value u. An error (abort) occurs if the datum field is non-zero.

`: Font \ pointsize "fontname" ++`
Sets the font used for this window. For example :

```
Font "MS Sans Serif"
```

`: bold \ -- ;`
The specified font is bold.

`: italic \ --`
Set the specified font to italic

`: strikeout \ --`
Adds the 'strikeout' attribute to the specified font.

`: underlined \ --`
Adds the 'underlined' attribute to the specified font.

`: End-Window \ --`
Ends a window definition. Used as part of the Window: .... End-Window construct.

`: Prop: \ n "name" ++ (compiling); -- addr (child)`
Create a WinProp field for the window currently being defined. The field is n bytes long and its address is returned when the word is executed during a window message handler.

`: Size \ w h --`
Sets the width and height fields of the window being defined.
The following words are compiled into the Forth vocabulary.

`: WinPropSize \ winstruct -- n ( 017 5 October 2005)`
Returns the size of the window properties for the given window *Winstruct* structure.

`: ClassPropSize \ winclass -- n ( 018 Jan-06)`
Returns the size of the window properties for the given window class *WinClass* structure.

Defining a Window

`: Window: \ WinClass "<name>" ++ (compile) ; -- addr`
The window defining word. A named dictionary entry is created consisting of a `WinStruct` data structure. The address of this data structure is set as the value of `CurrWinStruct` and the vocabulary `DefiningWindow` is added to the search order revealing the words to set/manipulate

the structure's fields. `End-Window` should be used to end the definition of this window and restore the search order.
Execution of the defined word returns the address of the `WinStruct` structure. A window is created using `Open-Window` which requires the handle of a parent window as an input parameter.

`: Open-Window \ hParent wstruct -- hWnd`
The word which creates window instance.

`: DefiningWindow? \ -- flag`
Returns TRUE if a window 'instance' is being defined. (Tests the value CurrWinStruct is non-zero.)

2.15 Handling WM_PAINT messages

To draw on a window in response to a WM_PAINT message, an application must first call the API function BeginPaint, which (amongst other things) fills in a PAINTSTRUCT and returns a handle to a device context on which to draw/paint. Any device context initialisation is next carried out before the application draws on the device context. When finished it must call EndPaint, after which the device context is no longer valid.

As an aid to this, the words `Start-Painting` and `Finish-Painting` are provided to replace the BeginPaint and EndPaint API calls respectively. These words handle the PAINTSTRUCT 'invisibly' and initialise the device context according to the settings in the WinProps structure.

The word `Open-DC` can be used to perform device context initialisation for a window at times other than in response to a WM_PAINT message. The decice context should be 'closed' using *\fo{Close-DC)

Device context initialisation involves the following:

• Set the DC font to that returned by WM_GETFONT .
• Set the brush to that specified in the winprop.Brush field of the WinProps structure.
• Set other DC properties specified in the WinProps structure which may be added from time to time.

`: Init-DC \ --`
Initialises the current device context (`CurrDC`) by setting the font of the DC to that returned in response to the current window's WM_GETFONT message.

`: Start-Painting \ --`
Calls the Windows API funtion BeginPaint using the current window parameters CurrWin etc. Places a handle to a device context in the <CurrDC> user variable and calls Init-DC. Since the PAINTSTRUCT is part of a window's WinProps, so there is no need for allocation etc.

`: Finish-Painting \ --`
The action to perform after START-PAINTING. Calls the Windows API function EndPaint and set the current DC to zero.

`: Open-DC \ --`
Readies the current window to be drawn on. Sets the CurrDC to the value returned by the API function `GetDC` and calls `Init-DC`. **NOTE:** This word is not for use when servicing a WM_PAINT message.

`: Close-DC  \ --`
'Closes' a device context after being 'opened' by `Open-DC`. CurrDC is set to zero.

`: Repaint   \ -- ;`
Repaint the current window using `Repaint-Window`.

2.16 A Real Example

```
\ The following is an example of creating a window using a class
\ procedure, an instance procedure and extended window properties.

\ The resulting window will keep a record of the number of times the
\ mouse has been clicked while pointing to the window, and will show
\ that number in the client area of the window.

NextID: IDM_Item10
NextID: IDM_Item11
NextID: IDM_Item12
NextID: IDM_Item13

NextID: IDM_Item20
NextID: IDM_Item21
NextID: IDM_Item22
NextID: IDM_Item23

Menu: Menu1
    IDM_Item10 SubMenu "&File"
        IDM_Item11 Item "&New"          highlight
        IDM_Item12 Item "&Open"
        IDM_Item13 Item "E&xit"
    end-menu

    IDM_Item20 SubMenu "&Edit"
        IDM_Item21 item "Cu&t"          greyed
        IDM_Item22 item "&Copy"         greyed
        IDM_Item23 item "&Paste"        greyed
    end-menu
end-menu


\ We start by defining a window class...

: Disp-Brush
\ with 'backlight'    196 242 96 RGB
    140 201 16 RGB
    CreateSolidBrush
;


WinClass: MyWindowClass

    CS_STANDARD Style
    IDI_ASTERISK Icon
```

```
    IDC_IBEAM Cursor

    cell    Prop: wp.Counter    \ count of mouse clicks
    200     Prop: wp.text       \ holds text to print.

    menu1 menu

End-WinClass

also windows

\ ... now define the actions which will take place

: BuildText     \ --
\ Used during the WM_PAINT handler to build the text required.
    s" Number of clicks is " wp.text place
    wp.Counter @ 0 <# #s #>  wp.text append
;

: PaintText     \ --
\ Used during the WM_PAINT handler to write the text. The Windows
\ API function TextOut is called. CurrDC is a word returning the
\ handle of the device context for the window which Start-Painting
\ created.
    CurrDC 2 2 wp.text count TextOut drop
;

: WClass-WMPaint    \ --
\ The action performed in response to a WM_PAINT message.
    Start-Painting
    BuildText
    PaintText
    Finish-Painting
;


: WClass-WMLButtonDown  \ --
\ The action performed in response to a WClass-WMLButtonDown message.
    wp.Counter incr
    CurrWin Null True InvalidateRect drop
;

\ now define the message chain for the window class..
\ [ClassMessages MyWindowClass
MyWindowClass [messages
    WM_Paint        does WClass-WMPaint
    WM_LBUTTONDOWN  does WClass-WMLButtonDown
messages]

\ ---------------------------------------
```

```
\ To add to this class behaviour, a window will be created
\ that limits the counter to less than 10 ..

: win-WMLButtonDown     \ -- ;
\ The action the window takes for a WM_LButtonDown message.
    inherited       \ do the class behaviour
    wp.Counter dup @ 10 mod swap !  \ 0<=counter<=9 OR counter in [0..9]
;


MyWindowClass WINDOW: MyWindow

    WS_VISIBLE Style
    WS_DLGFRAME Style+
    WS_SYSMENU Style+

    CW_USEDEFAULT CW_USEDEFAULT position
    200 100 position
    450 300 size
    Caption "My Window"

    [font
        name "courier new"
        16 size
        bold
        italic
        blue colour
\ variable pitch

    font]



    [brush
        cyan colour
\         solid
\        Vertical
    brush]
\ ))



End-Window

MyWindow [Messages
    WM_LBUTTONDOWN  does win-WMLButtonDown
messages]

: Go    \ --
    0 MyWindow Open-window set-currwin
;
```

```
EXTERN: int     PASCAL GetClassName( HWND hWnd, LPSTR lpClassName,int nMaxCount );
EXTERN: UINT    PASCAL  GetAtomName( ATOM nAtom, LPTSTR lpBuffer, int nSize );


256 buffer: 'buff
: ClassName     \ hwnd -- ca u
    'buff tuck 255 GetClassName    \ ca u
;

: attomname \ n -- ca u
    'buff tuck 255 GetAtomName
;

cr cr
.(  Type GO to see the window on screen) cr
.(  ----------------------------------) cr
cr
```

2.16.1 Useful Class Styles

CS_CLASSDC Allocates one device context to be shared by all windows in the class. One of the thread at a time successfully finishes a drawing operation.

CS_DBLCLKS Sends double-click messages to the window procedure when the user double-clicks the mouse while the cursor is within a window belonging to the class. CS_HREDRAW Redraws the entire window if a movement or size adjustment changes the width of the client area.

CS_NOCLOSE Disables the Close command on the System menu.

CS_OWNDC Allocates a unique device context for each window in the class.

CS_VREDRAW Redraws the entire window if a movement or size adjustment changes the height of the client area.

2.16.2 Less usefull class styles

CS_PARENTDC, CS_GLOBALCLASS, CS_SAVEBITS CS_BYTEALIGNCLIENT, CS_BYTEALIGNWINDOW

3 Dialog Boxes

This code provides a set of words which simplify many tasks associated with creating and managing dialog boxes. The differences between modal and modeless dialog boxes are largely hidden from application code for instance.

Dialog boxes are created as named dictionary items using a defining word and a "definition script" - words used to 'configure' a dialog box. A detailed explanation of the these and other words is given below.

The following code is a (very) simple example of code to create a dialog box called `BOX1` and display it. A more complex example may be found at the end of this document.

```
DialogBox: Box1
    Modeless
    Caption "Example Dialog Box"
    50 10 110 50 Position&Size
    WS_VISIBLE WS_CAPTION or Style
    WS_SYSMENU Style+
end-dialog

null Box1 drop
```

This code requires parts of the the windows creation code from Wins.fth, for instance the WinParams user variables (CurrhWnd etc) and WinProps.

3.1 Vocabularies

Two vocabularies are created to provide a limited amount of 'information hiding' - one contains the 'background' actions not used by applications, the other contains the scripting words which are used to create and define the various attibutes of a dialog box. Other high-level words are compiled into the Forth vocabulary.

`vocabulary DialogBoxes`
Contains the general 'internal' words used to create and manage dialog boxes. The 'high level' words which will usually be used by an application are compiled into Forth.

`vocabulary DefiningDialogs`
When a dialog box is defined this vocabulary is added to the search order. It holds words used to manipulate the dialog structure `/DialogDef` currently being defined.

3.2 Dialog box structures

The /DialogDef structure shown below holds information used to create a dialog box. It is an extension of the WinStruct structure and is used in a similar way when creating instances of a dialog.

```
struct /DialogDef
    /WinStruct +            \ now an extension to WinStruct
    0       field   dlg.X
    cell    field   dlg.Left
```

```
    0       field   dlg.Y
    cell    field   dlg.Top
    0       field   dlg.W
    cell    field   dlg.Width
    0       field   dlg.H
    cell    field   dlg.Height
    256     field   dlg.Caption     \ caption
\ 02Aug11    Cell    field   dlg.DlgProc     \ XT of a WinProc
    cell    field   dlg.Modeless    \ True or False
    cell    field   dlg.PointSize   \ for font
    cell    field   dlg.Weight      \ for font
    cell    field   dlg.Italic      \ for font
    129 chars field dlg.FontName    \ for font
    aligned
    cell    field   dlg.Menu        \ DB0004 Feb 2004
    0       field   dlg.Controls    \ start of any controls
end-struct
```

Manipulating the Dialog box structure

`0 value CurrDlgDef`
Holds the address of the dialog structure currently being defined. This value is used by the structure manipluating words such as `Style` and `Style+` etc.
The following words are contained in the `DefiningDialogs` vocabulary and are used to set/modify fields of the current dialog box, the address of which is held in `CurrDlgDef`. (See `DialogBox:` later.)

`: Style     \ u --`
Sets the dlg.Style field of the current dialog box structure to u. u is a set of bit flags defined by ORing WS_xxx and DS_xxx Windows style constants.

`: Style+    \ u --`
Adds the window/dialog box style value u to the current style by ORing the existing vlaue with u.

`: ExStyle   \ u --`
Sets the extended style of the dialog box currently being defined to u.

`: ExStyle+  \ u --`
Adds the extended style of the dialog box currently being defined to u.

`: Position      \ x y --`
Sets the position fields X and Y of the current dialog box structure. This will be the position of the dialog box when it is created.

`: Size          \ w h --`
Sets the width and height fields of the dialog box structure being defined. Note that these values usually indicate the size of the client area. See "A Note on dialog box sizes" below.

A Note on dialog box sizes

When an application creates a dialog box, it specifies its size (width and height) in dialog units. In typical Windows style it is not always clear what is meant by width and height!

How the width and height values are interpreted depend in part on the window styles assigned

the dialog box when created. The following observations have been made by me on two machines one running Windows 98, and the other Windows XP. I do not know of any other documentation regarding this.

In general, the width and height refer to the size of the client area. A border and/or a caption bar causes the creation of a larger window.

`: Position&Size \ x y w h --`
Combines the SIZE and POSITION operations into one.

`: Caption \ "caption" ++`
Takes the string following in the input stream and sets the caption field of the current dialog box structure. Uses GetPathSpec and so the string may be delimited by white space, the double quote character or parentheses.

`: | \ n1 "<name>" -- n1|n2`
The 'C' style bitwise OR operator. (The Forth word OR is available.) This word is state smart and allows the following style of coding:

```
WS_VISIBLE | WS_CLIPCHILDREN | WS_DLGFRAME Style
```

`: End-Dialog \ --`
Ends a Dialog box definition.

`: PointSize \ u --`
Sets the point size of the dialog box being defined to u. This will cause windows to attempt to adjust the font of the dialog box to have the given point size, when it creates the box.

`: Weight \ u --`
Sets the weight of the font used in the dialog box

`: Italic \ --`
Causes the dialog box font to be italics if possible.

`: Italics \ --`
Causes the dialog box font to be italics if possible.

`: Font \ pointsize weight italic "fontname" ++`
Combines the individual font words above into a single operation. When the dialog box is created, Windows attempts to set the font of the dialog box to that specified. For example :

```
10 100 0 Font "MS Sans Serif"
```

`: PropertySize \ u --`
Sets the size of the WinProps structure which is created when a dialog box is created. An ABORT occurs if the current property size greater than u.

`: Popup \ 'popup -- ;` `005 8/3/04`
'popup is the address of a popup menu (see the *Menus* documentation), which is displayed when a mouse pointer lies over the dialog box and the right button is pressed.

`: Menu \ addr -- ;` `DB0004 Feb 2004`
addr is the address of a menu template (see the MENUS documentation), and it is stored in the current dialog defintion structure. The menu is assiciated with the dialog box when it is first created.

`: Class \ winclass --`
Specifies the window class that is used to create the dialog box. This is useful if you wish to create dialog boxes which have special icons etc.

If the property size of the given window class is larger than the defauilt */WinProps*, the property size of the current dialog box is set to that value.

3.3 Handling Dialog Templates

Windows handles character strings in dialog boxes as Unicode characters. The following supplimentary words provide some provision for them.

`: UChars \ n1 -- n2`
Converts a number of Unicode characters to address units.

`: UChar+ \ add1 -- addr2`
Adds the number of address units of one unicode cahracter to add1 to give add2.

`: UChar! \ ch addr --`
Store the ANSI character ch at the address 'add' as a Unicode character.

`: uplace \ addr1 len addr2 --`
Places the ANSI character string at addr1 and len chars/bytes long, to the destination starting at the address addr2, as Unicode characters.

`: ucount \ caddr -- caddr len`
A version of COUNT for Unicode zero terminated strings. Returns the address of the first character and the length 'address units' (i.e. bytes). Add a UCHAR to skip over past the end.

`: u$. \ addr --`
Prints the Unicode counted string at addr to the console. (As $. but for Unicode a string.)
The structure that Windows requires to create a dialog box is known as a DLGTEMPLATEEX. It contains variable length and/or optional fields, dependant (for one thing), upon the dialog style.

Because of the variable/optional fields, the structure is created in memory while creating the dialog box. The dictionary structures are kept 'rigid'.

The first part of the Windows DLGTEMPLATEEX structure is fixed. It is duplicated here as DLGTEMPLATEEX(a) and extended in a 'variable length' way. The record structure fields are prefixed by *dte.* and in order to appear consistant the extended fields are accessed by words with the same prefix.

```
struct DLGTEMPLATEEX(a)
    _word   field   dte.Signature       \ has to be 1
    _word   field   dte.Version         \ must be $FFFF
    \ NOTE THAT the Win32 Programmers Help File specify
    \ the signature and version swapped.

    _dword  field   dte.HelpID
    _dword  field   dte.ExStyle
    _dword  field   dte.Style
    _word   field   dte.#DlgItems
    _word   field   dte.x
    _word   field   dte.y
    _word   field   dte.w
    _word   field   dte.h
end-struct
```

The next part of the DLGTEMPLATEEX 'structure' is as follows

1. Menu Identifier. It is one of the following, but in this implementation it will always be 16-bit zero.
   a. 16-bit zero : indicates no menu.
   b. 16-bit set to $FFFF, followed by 16-bit ID of menu in resource of exe file.
   c. null-terminated unicode string : name of menu in executables resource.

2. Window Class identifier. It is one of the following, but in this implementation it will always be 16-bit zero.
   a. 16-bit zero : indicates the predefined dialog class.
   b. 16-bit set to $FFFF, and 16-bit ID of a predefined system window class.
   c. null-terminated unicode string : name of window of a registered window class
3. Title - A null-terminated Unicode string, containing the title of the dialog box.
4. PointSize - 16 bits Specifies the point size of the font to use for the text in the dialog box and its controls. The pointsize, weight, Italic, and font members are present in an extended dialog box template only if the style member specifies the DS_SETFONT style.
5. Weight - 16 bits. Specifies the weight of the font in the range 0 through 1000. This can be any of the values listed for the lfWeight member of the LOGFONT structure. This member is present only if the style member specifies DS_SETFONT.
6. Italic - 16 bits. Indicates whether the font is italic. If this value is TRUE, the font is italic. This member is present only if the style member specifies DS_SETFONT.
7. Font. Specifies a null-terminated Unicode string that contains the name of the typeface for the font. This member is present only if the style member specifies DS_SETFONT.

The follwing words access the variable/optional fields:

```
: dte.MenuID    \ addr1 -- addr2
```
Returns the address of the MenuID field.

```
: dte.ClassID   \ addr1 -- addr2
```
Returns the address of the ClassID. In this implementation the MenuID field is always a 16-bit zero.

```
: dte.Title     \ addr1 -- addr2
```
Returns the address of the title field by 'jumping over' the ClassID field. The length of the jump depends on the contents of the field which is expected to be either a 16-bit zero or a zero-terminated 16-bit character string.

```
: dte.PointSize     \ addr1 -- addr2
```
Jumps over the title field to return the font point size field. The title field is a variable length 16-bit character zero-terminated unicode string.

```
: dte.Weight    \ addr1 -- addr2
```
Returns the font weight field.

```
: dte.Italic    \ addr1 -- addr2
```
Returns the font italic field.

```
: dte.Font \ addr1 -- addr2
```
Returns the font field.

`: .tmplt \ tmplt --`
For testing/debugging. lists the contents of a DLGTEMPLATEEX(a) structure.

`constant TemplateSize \ -- u ;`
The size used when allocating memory for a DLGTEMPLATEEX(a) structure.

`: Init-DialogTemplate \ 'template --`
Given the address of a dialog template structure, this word initialises its fields to pre-determined values. Used by the words to open a dialog box.

`: Str>Template \ caddr len tmplt --`
Copy the ASCII string "caddr len" to the dialog template 'Caption' field as a Unicode string.

`: Class>Template \ 'class tpmlt --`
Fills in the `dte.ClassID` field with the given class address (which may be zero).

`: DlgDef>template \ dlg tmplt --`
Take information contained in a dialog box structure, (see /DIALOGDEF), and transfer it to the windows dialog template structure.

3.4 The Tooltip control

A *Tooltip* is a Windows control which displays a line of text when the cursor hovers over an elected window or rectangular area of a window. Such windows and areas are said to be *tools*, and any number can be supported by a single tooltip control. A tooltip control is created with each dialog box in this code.

Each tool must be 'registered' with the tooltip and the word `Add-Tooltip` is provided to do this for a window or control with a dialog's tooltip control. `ShowsTip` can be used inside a dialog procedure instead. Tools as areas of a window are not supported at present.

There are numerous flags and settings used to add a tool to a tooltip control in Windows, but this code does not expose them. `Add-Tooltip` is the only word provided here to register a tool and is implemented such that a TTN_NEEDTEXT notification message to be sent to the dialog box (by the tooltip control) to request the text to show whenever a tool's tip is to be displayed. It is up to the application to provide that text.

The word `ShowsTip` is a wrapper around `Add-Tooltip` and assumes that the current window is a dialog box and takes the ID of a control as input.

Example code showing how to use tooltips is provided at the end of this section. It requires that the controls word set is loaded.

`: Create-Tooltips \ -- hTT`
Creates a ToolTip control using `CurrWin`) as the parent window, and returns its Windows handle. This word is called each time a dialog box is created.

`struct /ToolInfo`
The Windows TOOLINFO structure. Its fields are listed below.

```
cell    field   toolinfo.cbSize \ size of the structure
cell    field   toolinfo.uFlags \ bitflags
cell    field   toolinfo.hWnd   \ handle of window containing tool
cell    field   toolinfo.uId    \ ID or handle of tool
4 cells field   toolinfo.rect   \ bounding display rectangle
```

```
    cell    field   toolinfo.hinst  \ handle for string resource
    cell    field   toolinfo.lpszText \ text for the tool

: Add-ToolTip   \ hCtl hDlg --
```

The word an application uses to add a tooltip to a dialog box. *hCtl* is the handle to a control, *hDlg* is the handle of the dialog box. Controls in dialog boxes are not registered as tooltip controls by default. This is the word to do it.

```
: Del-ToolTip   \ hCtl hDlg --
```

Removes a control from a dialog's tooltip.
The following words provide additional help with tooltips.

```
: ShowsTip      \ ID --
```

Registers the control with the given ID as a tool in the current dialog's tooltip control.

```
: TipFor?   \ -- ID
```

Returns the ID of the control for which the tip text is required. This word should be used only while servicing a TTN_NEEDTEXT notification message.

```
: SetTip    \ z$ --
```

For use within a TTN_NEEDTEXT notification message handling routine, sets the text return value to the given string.

Example of using a dialog box's tooltip control.

Imagine a dialog box containing an edit box in which the user should enter his name and a button labelled "OK". The following code shows how to assign textual tips to these two controls.

```
NextID: IDC_NameBox     \ ID of edit box

\ We need a word to register those controls that we want to
\ show a tool tip. SetTippers below does this for both the
\ OK button and the edit box.

: SetTippers    \ --
    IDOK ShowsTip
    IDC_NameBox ShowsTip
;


\ The following wordS provides the text that will be shown
\ for a control by responding to the TTN_NEEDTEXT
\ notification message.

: ProvideTip    \ --
    case TipFor?
    IDOK        of  z" Click OK when finished"  SetTip  endof
    IDC_NameBox of  z" Type your name here"     SetTip  endof
    endcase
;

: Do-Notify     \ --
    case lParam nmhdr.Code @
    TTN_NEEDTEXT    of  ProvideTip      endof
    endcase
;

\ Finally the dialog box is defined, along with a word
\ to show it on screen.

DialogBox: EnterNameBox
    50 10 110 50 Position&Size
    Modal
    Caption "Tooltip Example"
    WS_VISIBLE WS_CAPTION or Style
    WS_SYSMENU Style+
    DS_CENTER Style+
    DS_MODALFRAME Style+
    DS_3DLOOK Style+
    DS_SetFont Style+
    10 100 0 Font "MS Sans Serif"

    5 5 100 12  IDC_NameBox     znull   EditBox
    35 30 40 12 IDOK            z" &Ok" DefPushButton

    WM_NOTIFY       does do-notify
    WM_INITDIALOG   does SetTippers
    WM_COMMAND      does:   wparamlo IDOK = if 0 Shut-CurrDlg then ;
end-dialog

: EnterName     \ --
    0 EnterNameBox drop
;
```

3.5 The Dialog box Procedure.

A single generic dialog box procedure is assigned to each dialog box. It does some initialisation work by setting up the WinProps for the dialog box, traverses the message chains to carry out the actions assigned to the various messages and releases WinProps memory when the dialog box closes.

There are two message chains for a dialog box: one defined by the application for that dialog box and another default chain which is common to each dialog box. See `DlgSw` below.

Applications can ignore how the dialog box is to terminate! Windows requires that dialog boxes are closed in different ways depending on whether the dialog box is Modal or Modeless. This is hidden within the generic dialog box procedure.

Below are listed the actions performed by the dialog procedure.

`Defer OPEN-ATTACHED(Dlg)  \ 'dlg --`
A word assigned to do DROP here, but will be re-vectored to create/open a list of controls which have been defined as children of the dialog. (See WinCtls.fth for details.) Called by `Init-CurrDlg` below as part of the WM_INITDIALOG message response.

`: Init-CurrDlg  \ -- ;`
Involked in response to the WM_INITDIALOG message and uses the Winproc parameter lParam as the address of a Forth `/DialogDef` structure. Information held in this structure is used to allocate memory for the dialogs WinProps and initialise some of its field values. Once this initialisation has been done, `OPEN-ATTACHED(Dlg)` is executed, which will create the controls to be contained in this dialog box.

`: Destroy-CurrDlg  \ --`
Executed by the generic dialog procedure in response to a WM_DESTROY message, this word removes the Window Properties which were set by Init-DialogBox. Since this word is called by the generic dialog box procedure, applications will not use it.
NOTE that the word forms part of the default message chain and so will not be executed before an application's WM_DESTROY message handler.

`: Close-CurrDlg  \ --`
Close the current dialog box. This word is executed in response to a WM_CLOSE message as part of the default message chain `DlgSw`.

The action taken here depends on whether the current dialog box is modal or modeless: for a modal dialog box the EndDialog API function is called, while for a modeless dialog box API function DestroyWindow is called. On entry to this word, if the value of RETURNVAL is non-zero no action is taken. In this way closing/destroying the dialog box can be controlled.
The dafault message chain is traversed after the dialog's message chain, and provides default processing for a number of messages.

```
MsgChain: DefDlgSw
    also windows

    WM_CONTEXTMENU  does do-popup
    WM_CLOSE        does Close-CurrDlg
    WM_DESTROY      does Destroy-CurrDlg
    WM_DRAWITEM     does wproc-DrawItem
```

```
    WM_MEASUREITEM  does wproc-MeasureItem
    WM_SIZE         does do-WM_SIZE
    WM_MOVE         does do-WM_MOVE
\ WM_SETFONT      does: ." Setting font for " currwin . cr ;
    WM_MOUSEMOVE    does wproc-WMMouseMove

    GWM-Popup       does    wproc-Popup

    GWM-SetOwnFont  does:   wparam wproc-OwnFont! ;
    GWM-GetOwnFont  does:   wproc-OwnFont? ReturnVal ! ;
    GWM-GetTextClr  does:   wproc-TextColour  ReturnVal !   ;
    GWM-SetTextClr  does:   wParam wproc-TextColour! ;

    GWM-SetOwnBrush does:   wParam wproc-OwnBrush! ;
    GWM-GetOwnBrush does:   wproc-OwnBrush? ReturnVal ! ;
    GWM-GetBrushClr does:   dproc-WinColour ReturnVal ! ;
    GWM-SetBrushClr does:   wParam wproc-WinColour! ;
    GWM-GetBrushHdl does:   dproc-WinBrush ReturnVal ! ;

    previous

;MsgChain
```

`4 1 callback: StdDialogProc \ hdlg mesg wparam lparam -- ior`
The common dialog procedure assigned to each dialog box. It performs the following operations:

1. Saves the dialog procedure parameters in their respective user variables using `Init-WndProc`.
2. Calls `Init-DialogBox` if the message is WM_INITDIALOG.
3. The message chains are scanned to perform any application-defined and default message handling.

3.6 Dialog Boxes with Class

A standard dialog box is based on a window class which is buried in Windows. It is possible to create a dialog box which is based on an application-defined window class and this section provides for this. A dialog box created in this way is referred to here as a *Class-dialog box* or *Class-dialog*.

A class-dialog box allows a class brush, class icon and class menu to be defined. (Although Windows seems to ignore class brushes). Through the message chain mechanism a class-dialog box exhibits similar 'inheritance' behaviour to a standard window.

A window class used for a dialogs has its own window procedure (i.e. not the generic window class procedure defined in Wins.fth). Like a standard window class it has its own message chain which (again like a standard window class) is linked to a default message chain.

A class-dialog has its own message chain, and like a window's message chain, it is linked to its window class. The message handler traverses the dialog's message chain first, followed by the class chain and then the default message chain. Finally the windows function `DefDlgProc` is called to finish the message handling.

`Inherited`, `-Inherited`, `-WinHandler` and `+WinHandler` can be used to alter the order of message chain traversal in exactly the same way as is done with a window. However this is not the model expected by Windows, and one or two tricks have been used to make it work as one might expect. To date no problems have arrisen with this approach.

Notice that the WinProps is created by the standard WM_INITDIALOG message handler - the only message which provides enough information to do so - and not by the WM_CREATE handler as is the case for the standard window class. And since the message chain's addresses are held in the winprops structure some messages will be missed. (WM_CREATE for example). An interesting strategy is to send each message which cannot be processed (such as WM_CREATE) back to a window's message buffer using the OS function `PostMessage`. This is not done yet, but can be if there is a need.

The Default Message Chain

The following words are used to form part of the default message chain for class dialogs.

`: defdc-EraseBkgnd \ --`
The action taken by default in response to a WM_ERASEBKGND message. The client area is filled with the colour specified as the background brush of the dialog class. This action can be avoided by executing `-Inherited`.

`: dlgclassMenu \ -- addr|0`
Returns the address of the menu specified in the current window ClassDef structure. Zero is returned if no menu is specified or the current window properties have not yet been created. A factor of `dcproc-Init`.

`: defdc-Init \ --`
Performed in response to the WM_INITDIALOG message. The WinProps have just been created and this word inspects them to create a menu if one was specified in the class.

`MsgChain: DefClsDlgChain`
The default message handler for class-dialog boxes. It has the following definition:

```
    also windows
\    WM_GETFONT      does do-GetFont
\    WM_GETFONT      does: cwp 0= if ."                zero props  !!!" cr then ;
\    WM_SETFONT      does do-SetFont
\ WM_SETFONT      does: ." Setting font for " currwin . cr ;

    WM_EraseBkgnd   does defdc-EraseBkgnd
    WM_InitDialog   does defdc-Init
    WM_DESTROY      does Destroy-CurrDlg
    WM_CLOSE        does Close-CurrDlg
    WM_CONTEXTMENU  does do-popup
    WM_DRAWITEM     does wproc-DrawItem
    WM_MEASUREITEM  does wproc-MeasureItem
    WM_SIZE         does do-WM_SIZE
    WM_MOVE         does do-WM_MOVE
    WM_MOUSEMOVE    does wproc-WMMouseMove


    GWM-SetOwnFont  does:   wparam wproc-OwnFont! ;
```

```
    GWM-GetOwnFont  does:   wproc-OwnFont? ReturnVal ! ;
    GWM-GetTextClr  does:   wproc-TextColour  ReturnVal !   ;
    GWM-SetTextClr  does:   wParam wproc-TextColour! ;

    GWM-SetOwnBrush does:   wParam wproc-OwnBrush! ;
    GWM-GetOwnBrush does:   wproc-OwnBrush? ReturnVal ! ;
    GWM-GetBrushClr does:   wproc-WinColour ReturnVal ! ;
    GWM-SetBrushClr does:   wParam wproc-WinColour! ;
    GWM-GetBrushHdl does:   wproc-WinBrush ReturnVal ! ;

    previous
;MsgChain
```

`: do-DefDlgProc \ --`

The word that calls the Windows function `DefDlgProc` and which is executed after all message chains have been traversed. No further action is taken if the value returned by `DefDlgProc` is zero, but any other value replaces the value held in `ReturnVal` and so becomes the message 'result'.

`4 1 callback: ClassDlg-ClasProc`

The name of the window procedure for all dialog window classes. It has three main actions:

- Initialisation : set the various user variables using `Init-WndProc` and `IsWinHandler`.
- If the message is WM_INITDIALOG create the dialog box WinProps by executing `Init-CurrDlg`.
- Execute the message handling routine(s) assigned to the message chains. These are in order:
  1. The dialog box message chain,
  2. The dialog box class message chain,
  3. The class-dialog default message chain `DefClsDlgChain`, and finally
  4. the windows message handler is called through `do-DefDlgProc`

`4 1 callback: ClassDlg-Proc \ hdlg msg wparam lparam -- 0`

The dialog box procedure for all class-dialog boxes. Its is essentially a *noop*, and simply returns zero.

`: DlgClass:     \ "<name>" --`

Creates a window class on which a dialog box can be based. This word is used in exactly the same way as `Winclass:` is used (See the chapter on creating windows and classes).

Notes

All the standard WinClass properties can be specified in a dialog class, but note the following.

- If a class menu is specified, it is created in the default message chain `DefClsDlgChain`, that is AFTER the dialog and dialog class message chains actions. However, any specified dialog box menu has already been created by this time (by `Init-CurrDlg`), and in this case the class menu is NOT created.
- If a valid class icon is specified, it is shown on all class-dialog boxes belonging to that class. If no icon is specified, the Windows logo is shown on all those dialog boxes which do not have the DS_MODALFRAME style, in which case no icon is displayed. (This is a feature of Windows not the GUIgen code).

- A background brush can be specified. Windows seems not to use it to erase the client area. This code intercepts the WM_ERASEBKGND message in the default message chain `DefClsDlgChain` to erase the dialog box with the correct brush.
- Notice that the extra window space field (WndClassEx.cbwndextra) is set to 30 - dialog boxes use the space for "housekeeping". Applications requiring extra window space will need to take account of this.
- As in the case of the normal use of window classes, beware of any WinProps used by the dialog class: Application dialog boxes must build further properties "on top" of these. Use `ClassPropSize` (DlgClass – propsize) to find a class's property size.

3.7 Creating Dialog Boxes

```
: ShowModal     \ hParent 'dlg -- RetVal
```

Creates and displays a dialog box as a *modal* box with the given owner.

```
: ShowModeless      \ hParent 'dlg -- hWnd|0
```

Creates and displays a dialog box as a *modeless* box with the given owner.

```
: Init-CurrDlgDef   \ --
```

Used by `DialogBox:` when a dialog box structure is created in the dictionary. This word sets initial/default field values.

```
: DefiningDlg?      \ -- flag
```

Returns TRUE to indicate that a dialog box is being defined. Used by the window control words while compiling controls into a dialog box definition.

```
: Dialog:       \ "name" ++ addr
```

Creates a named dictionary entry and allots a DialogDef structure. This starts a dialog box definition which will be finished by END-DIALOG. Execution of the child word places the address of the structure on the stack. Use either ShowModal or ShowModeless to display the dialog box.

```
: Shut-Dialog   \ n hDlg --
```

Close the dialog box whose handle is hDlg. If the dialog box is modal, the value n is returned to the application.

```
: Shut-CurrDlg  \ n --
```

Perform `Shut-Dialog` using the current window handle as the dialog box handle.

3.8 A Simple Drawing Example

The code shown below defines a dialog box upon which the mouse can be used to draw shapes by depressing the left mouse button.

```
\ Loading required words and API calls.
\ -------------------------------------

EXTERN: BOOL    PASCAL ReleaseCapture(VOID);
EXTERN: HWND    PASCAL SetCapture( HWND hWnd );
Extern: Bool    PASCAL MoveToEx( HDC, int X, int Y, LPPOINT lpPoint)
Extern: Bool    PASCAL LineTo( HDC, int X, int Y)
```

```
\ Defining the dialog box
\ -----------------------

\ Create an extended WinProps structure. Space for it will be allocated
\ when the dialog box is created.
\ There are three fields in this structure : Drawing holds a flag which
\ is set TRUE to indicate that the pen is drawing. The X and Y fields
\ store the last recorded mouse coordinates.

Struct /DrawTestProps
    /WinProps +
    cell    field   drawtest.Drawing
    cell    field   drawtest.x
    cell    field   drawtest.y
end-struct

Dialog: DrawTestBox
    /DrawTestProps PropertySize
    50 10 110 50 Position&Size
    Caption "Detecting Mouse Movements"
    WS_VISIBLE WS_CAPTION or Style
    WS_SYSMENU Style+
    DS_CENTER Style+
    DS_MODALFRAME Style+
    DS_3DLOOK Style+
    DS_SetFont Style+

    10 100 0 Font "MS Sans Serif"

    [brush
        255 255 128 rgb colour
    brush]


    32 prop: dlgfld


end-dialog

\ The word DrawTest below, will create the dialog box and show it on
\ screen. It is 'forced' to be a Modeless box which allows the Forth
\ Console window to be used to issue commands. The returned value is
\ the dialog handle. Setting this as the current window allows one to
\ test the dialog procedure functions.

: DrawTest  \ --
    null DrawTestBox ShowModeless   \ hDlg ; open dialog, handle returned
    Set-CurrWin                     \ -- ; make this the current window
;

: Draw      \ --
```

```
\ DRAW is the MODAL version. The Forth console cannot
\ be used while in this mode.
    null DrawTestBox ShowModal drop \ drop any return value
;

\ the dialog's message handlers
\ ----------------------------

: do-LeftMouseDown      \ -- ; performed in response to WM_LBUTTONDOWN
    cwp drawtest.Drawing @ 0=
    if  cwp drawtest.Drawing on
        CurrWin SetCapture drop
        lParamLo cwp drawtest.x !
        lParamHi cwp drawtest.y !
    then
;

: do-LeftMouseUp    \ -- ; performed in response to WM_LBUTTONUP
    cwp drawtest.Drawing @
    if  cwp drawtest.Drawing off
        ReleaseCapture drop
    then
;

: w>s   \ 16b -- 32b
    dup $8000 and if $FFFF0000 or then
;

: do-MouseMove  \ -- ;  performed in response to WM_MOUSEMOVE
    cwp drawtest.Drawing @
    if  CurrWin GetDC                       \ hDC
        dup GetStockObject                  \ hDC obj
        over swap SelectObject drop         \ hDC
        dup cwp drawtest.x @                \ hdc hdc x
        cwp drawtest.y @ null MovetoEx drop \ hdc
        lParamLo w>s
        lParamHi w>s                  \ hDC x y ; current position

        3dup LineTo drop                    \ hDC x y ; draw line
        cwp drawtest.y ! cwp drawtest.x !   \ hDC ; save position
        CurrWin swap ReleaseDC drop         \ --
    then
;

[DlgMessages DrawTestBox
    WM_LBUTTONDOWN  does do-LeftMouseDown
    WM_LBUTTONUP    does do-LeftMouseUp
    WM_MOUSEMOVE    does do-MouseMove
messages]


\   How to use:
```

```
\       DRAWTEST    to test the dialog box
\       DRAW        to run it as a modal box - the 'finished thing'
```

---

4 Window Controls

4.1 Introduction

This wordset provides a way to use of any Window Control and allows none-default behaviour to be added to a control easily.

Windows registers the so-called *predefined* controls (i.e. *button*, *combobox*, *edit*, *listbox*, *scrollbar* and *static* controls), but the Windows function `InitCommonControls` must be called to register the *common controls*. (See the VFX manual for more details about loading Window libraries).

By registering new window classes, the words here allow an application to create unique customised controls.

How to use this code

A control must first be 'defined' (i.e. a Forth word created), before it can be used in an application. The control definition consists of a data structure which holds some details about the control - most usually its style.

As an example, suppose an application required an 'Edit' control to be shown on a dialog box. A suitable definition might be :

```
WinControl: Line-EditBox "edit"
    WS_CONTROL ES_LEFT or Style
    WS_EX_CLIENTEDGE ExStyle+
end-Control
```

The above code, creates the word LINE-EDITBOX in the dictionary : a WinControl data structure specifying an edit control and the stated styles.

Once this control has been defined, it may be used any number of times to create an 'edit' control with those styles and a default behaviour. It will most often be used to specify controls as being child windows of dialog boxes within the dialog box definition.

For example a typical use of Line-EditBox within a dialog box is as follows :

```
DialogBox:  TextDlg
    50 10 110 50 Position&Size
    Modal
    ... other code ...
    5 5 100 10 IDC_EditBox z" Hello"   Line-EditBox
end-dialog
```

Controls can also be "embedded" in window class definitions and window definitions in a similar way. Controls can also be created 'in code'. For instance the line

```
5 5 100 10 IDC_EditBox z" Hello"   Line-EditBox open-control drop
```

if part of a WM_INITDIALOG message handler has a similar efect to the example above.

For cases where the supplied controls are insufficient there is the ability to create brand new ones. A window class is constructed containing the desired basic behaviour (using `WinClass:`) and an *application control* created based on that class using `AppControl:`.

Since a control can be constructed from a standard window, and a standard window can contain controls, it is possible to create 'super controls' or 'compound controls'. A possible example is a Button Panel - a single control which contains a number of buttons each of which can be assigned a different action.

4.1.1 How it works - Subclassing controls.

Windows allows subclassing of any window (including controls) by assigning a new callback routine to become the new WndProc for that window. The application must remember the previous/old callback address and call that routine to inherit previous actions.

In this scheme each control defined (by WinControl:) is subclassed when created. The new procedure (the same for each control), runs the original procedure by default AFTER any message handlers assigned to the control (by either the word `DOES` or `DOES:`).

In other words, once a control has been created, the message handler for that control is as follows:

• Windows calls the procedure `WinControlProc` (defined below)
• which performs three actions:

1. The message parameters are saved in user variables. (Exactly like the standard window procedure - Wins.fth).
2. The controls message chains are traversed and any message specific actions performed.
3. Finally, the original window procedure for that control is called.

The benefits of this scheme include the following:

• Message handlers are Forth words which have no net stack effect. One word assigned to a single message.
• Windows *callback* routines are eliminated from application code.
• Default message processing is performed automatically - an application must explicitly state when NOT to do it.

4.1.2 Control Properties

Each control is assigned/allocated a small block of memory to hold certain data (for instance the controls message chain is stored there). The memory is a WinProps structure (see Wins.fth) and may be extended for any control to provide for 'personalised' behaviour.

4.1.3 Other considerations.

It is important to note that no messages will be 'detected' by a control until its WinProps has been created. For instance WM_CREATE will be missed.

For this reason, a new message has been 'invented' : GWM_INITCONTROL is sent to the control immediately after the control is created (by the Windows API function CreateWindowEx). This gives the application the chance to initialise any data structures and create child controls etc.

4.2 WinControls code

4.2.1 Introduction

The vocabulary `WinControls`, is used to contain nearly all words in order to 'hide' implementaion details. The exceptions are the 'high-level' words such as `WinControl:` and `End-Control`.

The vocabulary `DefiningControls` holds words used when defining a WinControl and is added to the search order by the words `WinControl:` and `AppControl:`.

`vocabulary Wincontrols`
The vocabulary which houses most of the WinControl words.

`vocabulary DefiningControls`
The vocabulary holding words used to define/describe a control.

`Variable 1st-Control`
Holds the address of the last control to be defined and is the first control in the list of all controls. Its sole use is by `.Controls` - a debug/testing aid.

4.2.2 The WinControl structure.

/WinControl is an extension of the WinStruct structure (used to define standard windows), used to store information at compile-time which will be used at run-time.

```
struct /WinControl
    /WinStruct +            \ this is an 'extended' WinStruct
    64      Field   WinControl.ClassName \ class name
\ 02Aug11    cell    Field   WinControl.ProcXT   \ now obsolete
\ 22Sep11    cell    field   WinControl.OrigProc \ original winproc
    cell    Field   WinControl.Next     \ next one
    cell    field   WinControl.FirstCtl \ address of 1st control
end-struct
```

- WinControl.ClassName holds the name of the Window class as a zero-terminated string.
- WinControl.Next holds the address of the next control in a linked list of all controls created. This field is accessed by .CONTROLS to show a list of all controls created.

4.2.3 Manipulating a WinControl structure

WinControl data manipulation words are compiled into the `WinControls` vocabulary which becomes the current vocabulary when a control is created. The value `CurrWinCtl`, which holds the address of the control structure being defined (i.e. a WinControl structure), is used by those words to set and adjust the values held in the structure.

```
8 cells buffer: CWC
```
A 'stack' of window controls being defined. No overflow or underflow is detected - each condition results in 'wrap araound' one way or the other.

```
7 cells value cwcoff
```
Offset into the CWC stack. This is the offest to the current item on top of the CWC stack.

```
: >cwc      \ addr --
```
Push the given address onto the CWC stack.

```
: cwc-      \ --
```
Make the top item on the cwc stack zero and the drop it from the stack. Seting the value to zero allows a test for zero when laying down controls. (see `DefiningCtl?` later).

```
: CurrWinCtl    \ -- add
```
Return the address on top of the cwc stack without popping it.

```
0 value GroupState
```
GroupState is assigned a value of 2, to indicate that the control will be the start of a group. A value of 1 indicates part of a group. A zero value indicates not in a group. This value is manipulated by words such as `[GROUP`, and `GROUP]`.
The following words are compiled into the `DEFININGCONTROLS` vocabulary, and are for use inside a `WINCONTROL:` ... `END-CONTROL` definition.

```
: Style     \ u --
```
Sets the current control's style to u.

```
: ExStyle   \ u --
```
Sets the current control's extended style to u

```
: Style+    \ u --
```
Adds u to the current control's style using OR.

```
: Style-    \ u --
```
Remove the bit flags set in u from the current control's style using AND with the inverse of u.

```
: ExStyle+  \ u --
```
Adds u to the current control's ExStyle using OR.

```
: ExStyle-  \ u --
```
Remove the bit flags set in u from the current control's ExStyle using AND with the inverse of u.

```
: PropertySize      \ n --
```
Sets the WinProps size to the value n. When the control is created a WinProps structure of size n is allocated.

```
: End-Control       \ --
```
Ends a WinControl definition.

```
: Popup     \ 'popup --
```
Sets the current control's popup menu. 'popup is the address of a popup menu defined by POPUP:

```
: Datum     \ u --                    [0014 Apr 04]
```
Sets the WinStruct.Datum field of the current control to the value u. An error (abort) occurs if the datum field is already non-zero.

```
: SetMiscLo \ n -- ;
```

Sets the WinStruct.MiscLo field of the current control to the value u. The value u will later be passed to the control's winprop.MiscLo field.

`: SetMiscHi \ n -- ;`
Sets the WinStruct.MiscHi field of the current control to the value u. The value u will later be passed to the control's winprop.MiscHi field.

`: GGStyle+ \ u --`
Adds the bit of u to the WinStruct.GGStyle field of the current control using logical OR.

```
Example
    WinControl: EditBox "edit"
        WS_CONTROL ES_LEFT or Style
        WS_EX_CLIENTEDGE ExStyle+
    end-Control
```

4.2.4 Exposed words to manipulate controls

The Following words are used outside a WinControl definition to manipulate control properties.

`: [Group \ --`
Used within a dialog box definition, to mark the start of a control group.

`: In-Group \ --`
Ensures the next control specified in a dialog box is part of the previous group of controls.

Note: This word is used internally - there is no need for its use in application code.

`: Group] \ --`
Marks the end of a control group within a dialog box definition.

```
Example
DialogBox: MyDlg
    Modeless
    WS_VISIBLE WS_CAPTION or WS_SYSMENU or Style
    DS_CENTER DS_MODALFRAME or DS_3DLOOK or Style+
    0 0 145 120 Position&Size
    Caption "Password Generator"
    10 100 0 FONT "MS Sans Serif"
    [Group
      5 22 90 40  IDC_STATIC 1- z" Date & Time" GroupBox
      10 35 30 10 IDC_STATIC 2 - z" Date"       LTEXT
      10 45 30 12 IDC_Date   z" "              EDITBOX
      50 35 40 12 IDC_STATIC 3 - z" Time"       LTEXT
      50 45 30 12 IDC_Time   z" "               ViewBox
    group]
....
```

4.2.5 The sub-classing of controls

Under Windows the term 'subclassing' is used to indicate the process of assigning a different

window procedure to a window instance. In the process the address of the previous procedure is remembered and later called by the new procedure if and when required . In this way a window aquires new behaviour in addition its old behaviour.

GUIgen controls are ALL subclassed - each control being assigned the same procedure (See `WinControlProc` later). The following words provide the subclassing function. The words are used 'internally' and would not normally be used by an application.

`Create "OrigWinProc"  z", OrigWinProc"`
The name of the Window Property the value of which is the address of the original window procedure for that window/control.

`: Subclass-Window  \ 'callback hWnd --`
'Subclass' a window. That is, set a new window procedure for the given window. The original procedure is stored in the window property "OrigWinProc".

`: -Subclass-Window  \ hWnd --`
Remove the window procedure set by SUBCLASS-WINDOW, and replace it with OrigWinProc. The OrigWinProc property is removed. This word is executed when a control is destroyed.

`: Do-OrigCtlProc  \ --`
Perform the original/default window processing for the 'current' subclassed control (i.e. the control whose control procedure is now being executed).

As of March 2006, this word should not be used by applications, which should use the message chain words (i.e. `-inherited` and friends).

4.2.6 Setting up the control's extended WinProps

Just like standard windows and dialog boxes, controls (under GUIgen) each have their own WinProps data, and memory for this data structure is allocated as soon as the control has been created. But like dialog boxes, the early messages are missed by GUIgen - subclassing can only occur after a control has been created and WM_CREATE is missed (as are a few others).

Because the WM_CREATE message is missed, an *initialise* message is sent as soon as each control has been created. This is `GWM_InitControl` and applications can use it to perform its own initialisation actions.

`AppMessage: GWM_InitControl`
An 'application defined' message number. This message is sent to each control immediately after it has been subclassed. Applications cannot 'see' a control's WM_CREATE message - hence the need for this message. (An application's own controls, defined using `AppControl:`, receive all messages).

`AppMessage: GWM_CreateControls`
An internal message which is sent to a control immediatly before the GWM_InitControl message and is used to create any attached controls. It is not expected that applications will trap this message.

`: Create-CtlProps  \ hCtl 'ctl -- ;`
Create the WinProps for this control. This word distinguishes between Windows supplied controls and application controls defined using `AppControl:` whose WinProps have already been allocated by the time this word is executed. (Application controls create WinProps in response to the WM_CREATE message).

```
: Setup-CtlProps    \ hCtl 'ctl -- ;
```
Setup/initialise the winprops for the current control. Calls CREATE-CTLPROPS and sets control-specific data values.

4.3 The control window procedure

Each control is subclassed with the same callback function called WinControlProc, which performs some default processing, executes the control-specific actions defined by the application, and calls the original window procedure. Default actions are defined for the messages WM_DRAWITEM WM_CONTEXTMENU and WM_DESTROY.

```
: Destroy-CurrWinControl  \ --      ;       0016 July 2004
```
Called in response to a WM_DESTROY message of a control by `WinControlProc`. The control's window properties are deleted by this routine and the sublass routine is removed. It forms part of the default message chain. Note that after this word is executed (perhaps by executing `inherited` in an application message handler), the value returned by `CWP` is zero - the allocated memory has been freed.

```
defer Open-Attached(Ctl)
```
The word which will create any attached controls

```
: Make-ControlControls  \ --
```
The handler for the internal GWM_CreateControls messages. Calls the code to create any controls attached to this one.

```
MsgChain: DefCtrlSwitch
```
A message chain which is used to perform some default processing:

```
    WM_DESTROY          does Destroy-CurrWinControl
    WM_DRAWITEM         does wproc-DrawItem
    WM_MEASUREITEM      does wproc-MeasureItem
    WM_CONTEXTMENU      does do-popup
    GWM_CreateControls  does Make-ControlControls

\    CN_CTLCOLORMSGBOX       does wproc-CN-CTLCOLORxxx
    CN_CTLCOLOREDIT         does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORLISTBOX      does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORBTN          does wproc-CN-CTLCOLORxxx
\    CN_CTLCOLORDLG          does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORSCROLLBAR    does wproc-CN-CTLCOLORxxx
    CN_CTLCOLORSTATIC       does wproc-CN-CTLCOLORxxx
    WM_MOUSEMOVE            does wproc-WMMouseMove

    GWM-SetOwnFont  does:   wparam wproc-OwnFont! -inherited ;
    GWM-GetOwnFont  does:   wproc-OwnFont? ReturnVal ! -inherited ;
    GWM-GetTextClr  does:   wproc-TextColour  ReturnVal ! -inherited ;
    GWM-SetTextClr  does:   wParam wproc-TextColour! -inherited ;

    GWM-SetOwnBrush does:   wParam wproc-OwnBrush! -inherited ;
    GWM-GetOwnBrush does:   wproc-OwnBrush? ReturnVal ! -inherited ;
    GWM-GetBrushClr does:   wproc-WinColour ReturnVal ! -inherited ;
    GWM-SetBrushClr does:   wParam wproc-WinColour! -inherited ;
    GWM-GetBrushHdl does:   wproc-WinBrush ReturnVal ! -inherited ;
```

```
4 1 callback: WinControlProc     \ hdlg mesg wparam lparam -- ior
```
The procedure given for each subclassed control. The following actions are taken :

a. WinParams (user variables) are initialised.
b. Message handling is performed in the following order:
    1. The control message chain is traversed, then
    2. `DefCtrlSwitch` is traversed, and finally
    3. the original window procedure for the control is called.
c. Finally the value held in `ReturnVal` is returned as *ior*.

```
: Setup-SubClassing     \ hCtl 'Ctl --
```
hCtl is the handle of a (newly created) control, *'Ctl* is the address of a control (a /WinControl). The word does three things:

- subclasses the control by calling `SubClass-Window`,
- creates the control's WinProps with `Setup-CtlProps` and
- sends a GWM_INITCONTROL message to the control.

4.4 Some miscellaneous words

```
: Modify-Style  \ style -- style'
```
Add or remove the WS_GROUP style if necessary according to the value of GroupState. Used immediately prior to opening a control.

4.5 Mapping base units to pixels...

```
: CharSize      \ hWnd -- w h
```
Given a valid window handle, calls the GetTextExtentPoint32 API function to calculate the average width and height of a character.

```
: ChWidth   \ hWnd -- w
```
Given a valid window handle, calls the GetTextExtentPoint32 API function to calculate the average width of a character.

```
: ChHeight  \ hWnd -- w
```
Given a valid window handle, calls the GetTextExtentPoint32 API function to calculate the height of a character.
A *character unit* is 1/4 of the average character width and 1/8 of its height just like dialog base units. On this basis are the following words `CharUnits>Pixels` and `MapWinDims` defined. The word `MapDlgDims` uses the Windows API funtion `MapDialogRect` which is documented to perform a similar action.

```
: CharUnits>Pixels  \ x y chwidth chheight -- x' y'
```
Convert the character units x and y into pixels (x' and y') given the average width and height of a character.

```
: MapDims   \ x y w h -- x' y' w' h'
```
Convert character unit dimensions into pixel dimensions for the current window. This word is called when creating a control with `open-dialog`. The current winprops field `winprop.WinType` is inspected in order to decide which mapping function to use.

```
: hUnits    \ n1 -- n2
```
Convert n1 horizontal base units to pixels

```
: vUnits    \ n1 -- n2
```

Convert n1 vertical base units to pixels
The following words position controls within a window. The following notation is used in the stack effect comments:

- *id1* is the ID of a control to be positioned.
- *id2* is the ID of a control relative to which ID1 is positioned.
- *hUnits* is the number of horizontal base units
- *vUnits* is the number of vertical base units

```
: above     \ id1 id2 vunits --
```
Position *id1* above *id2* such that there is a gap of *vUnits* between them.

```
: below     \ id1 id2 vunits --
```
Position *id1* below *id2* such that there is a gap of *vUnits* between them.

4.6 Opening Controls

In order to act in a way similar to Windows, a control's position and dimensions are expressed in *dialog units* and must be converted to screen units (i.e. pixels) before being created. Windows does this using the function `MapDialogRect` and this code does the same.

A dificulty arises when creating controls in non-dialog windows since `MapDialogRect` works only for dialog boxes. To accomodate this, the word `MapDims` (above) is used, which executes code according to whether the current window handle (i.e. `CurrWin`) belongs to a dialog box or not. (**Note:** this is a change to the way controls were opened before April 2006 when the two words *Open-Control* and *Open-DlgControl* were supplied).

The following words used to create (or *open*) a control are listed below. Stack parameters are all single-cell with the following meaingings:

- *'ctl* - the address of a /WinControl structure.
- *x, y* - position in base units.
- *w, h* - width and height in base units.
- *Textz* - the address a zero-terminated string.
- *hWnd* - the address if a window or control.

```
: Get-ClassName     \ 'ctl -- Textz
```
Get the class name on which this control is based. If the class is an application defined class, that class is registered with windows first, and the address of the name within the class structure is returned. Otherwise the address of the name in the control structure is returned.

```
: (Open-Control)    \ x y w h ID Textz 'ctrl -- hWnd
```
Open/create a new window control by calling the API function CreateWindowEx, specifying the current window (`CurrWin`) as the parent and GetModuleHandle is used to specifiy the hInstance parameter. This word which is called by `Open-Control`, would not usually be called any other time.

```
: Open-Control  \ x y w h ID Textz ctl -- hWnd
```
Create a GUIgen control. *hWnd* is the handle to the control which has been subclassed, its WinProps have been created and a GWM_InitControl has been sent to its parent.

4.7 Laying down control info into the dictionary

Both windows and dialog boxes may be defined as 'containing' controls by specifying a position, size and other data in strict order. These data are comma-compiled into the dictionary as a singly linked list of items, and then accessed when the dialog box or window is opened - each control being created as child windows. Controls are usually specified at the end of the window or dialog description.

The following structure is compiled for each contol :

```
struct  InlineCtl
    _word   field   InlineCtl.X     \ position of LH side
    _word   field   InlineCtl.Y     \ position of top
    _word   field   InlineCtl.W     \ width
    _word   field   InlineCtl.H     \ height
    _word   field   InlineCtl.ID    \ identifying number
    aligned
    0       field   InlineCtl.Win   \ address of window
    cell    field   InlineCtl.CTL   \ or control def
    Cell    field   InlineCtl.Next  \ address of next InlineCtl
    Cell    field   InlineCtl.Group \ holds the group state flag
    0       field   InlineCtl.Text  \ variable length z-string
end-struct
```

`: CtlSpec,      \ x y w h ID Textz ctl GpFlag --`
Compiles the given information into an InlineCtl structure at here in the dictionary. A factor of the word `Lay-Control`.

`: CtlSpec@      \ addr -- x y w h ID Textz ctl GpFlag`
Retrieves control information from an InlineCtl structure which lies at the given address. Used by the words which open/create the controls 'embeded' in dialog boxes and windows.

`: CreateCtl     \ addr --`
Given the address of an InlineCtl structure, retrieves the information contained, and opens/creates the control using `Open-Control`.

`: '1stCtl   \ -- addr`
Returns the address of the first control in the current WinClass:, Window: or DialogBox: definition.

`: IsNextCtl     \ addr -- ;`
Mark the address given as being the next control in the control list of the current WinClass:, Window: or DialogBox: definition.

`: Lay-Control       \ x y w h ID Textz Ctl --`
'Comma compile' the control information given along with the value held by GroupState. Uses CtlSpec, and data will be retrieved using CtlSpec@ when the dialog box or window is opened.

`: (OPEN-ATTACHED-dlg)       \ 'dlg --`
The action assigned to the deferred word OPEN-ATTACHED(dlg) which is executed after a dialog box is opened. Its action is to open each control compiled into the dialog box definition and its dialog class if it has one. Each control is accessed in the order in which they were compiled into the dictionary.

`: (OPEN-ATTACHED-win)       \ 'WinStruct --`

Given the address of a WinStruct structure, each InlineCtl structure attached is opened using CTL-IN-WIN. This is the word assigned to OPEN-ATTACHED(Win) which is executed after a WinClass instance is created (during the WM_CREATE message). Any controls specified in the defining window class are created first, then any controls attached to the given winstruct.

```
: (open-attached-ctl)   \ 'ctl --
```

Creates any controls attached to the given control structure.

4.8 Defining Controls

Three control creating words are supplied: WINCONTROL:, APPCONTROL: and ANOTHER. Each creates a /WINCONTROL structure with particular styles and attributes. WINCONTROL: builds a standard Windows-supplied control, APPCONTROL: builds an application-defined control based on a window class defined using WINCLASS:. ANOTHER creates a copy of an existing control allowing similar controls to be built with little duplication of effort.

The following words are used to create controls.

```
: DoControl     \ ... ctl -- ???
```

This word provides the DOES> part of the control defining words `WinControl:`, `AppControl:` and `ANOTHER`. Its action is complicated - there are THREE states to consider. In each case the following form of notation is expected:

```
x y w h ID z" text" ControlName
```

If STATE is true we are in a colon-definition and the top item on stack is compiled as a cell literal. Typically the phrase above would be followed by `"open-control drop"`

If STATE is FALSE there are 7 items on the stack (i.e. position, size, ID, string adress and addres of a control). If a window class, a window or a dialog box is being defined the 7 values are compiled into the dictionaary using `Lay-Control`.

Otherwise, we are interpreting but NOT defining a window class, a window or a dialog box. In this case no further action is taken.

This messy behaviour allows controls to be compiled into dialogs, windows and window classes implicitley, while also allowing the use of controls within word definitions and interpretively.

```
: Build-Control     \ --
```

The main compile-time action of the defining words WINCONTROL:, APPCONTROL: and ANOTHER.

Create a control structure at HERE in the dictionary, the address of which is stored in the value CURRWINCTL, and is linked in to the chain of controls. The vocabulary DEFININGCONTROLS is added to the search order to be removed later by END-CONTROL.

```
: WinControl:   \ "name" "classname" ++ ; x y w h ID z$ -- hWnd
```

Create a word called 'name' in the dictionary, which has an "attached" /WinControl data structure in which the field WinControl.ClassName contains the zero-terminated string 'classname'. Execution of the child word runs DOCONTROL which either compiles code to open the window at 'run-time', or leaves its address on the stack if interpreting. If a window class,

a window or a dialog box is being defined then the control is compiled into that data structure (See DOCONTROL)

```
: .Controls     \ --
```

Print a list of the names of defined WinControls.

4.8.1 Application controls and *Another* controls.

An *Application Control* is a control based on an application-defined window class. Typically the window class will define most of an application control's behaviour with minor additions/changes being defined for the control itself. For instance a text entry application control class might accept any typable character with different controls rejecting certain key characters and allowing others.

An application control is defined using `AppControl:` then follows the `WinControl:`-style definition script ending with the word `end-control`.

Another controls are copies of other controls each one defined with the word `Another` followed by the standard control definition script ended by `end-control`.

```
: AppControl:   \ 'winclass "<name>" ++
```

Create a word called 'name' in the dictionary, containing a /WinControl data structure in which the WinStruct.ClassDef field is set to the given value 'winclass . Child words are immediate and perform DOCONTROL when executed. See below for a typical example of use.

```
: Another       \ "<controlname>" "<name>" ++
```

Create control in the dictionary called <name> which is an exact copy of the control <controlname>. The vocabulary search order is set to allow the new control's fields to be modified (e.g. using STYLE, STYLE+). This is a short cut to defining a similar control, and must end with END-CONTROL.

Example of the use of AppControl: and Another

```
The Following example builds a simple control which accepts
keyboard input and adds each character typed to its own
text. The control is called TextEntry.

Another is then used to create a control called
DigitEntry which is identical accept that only digit
characters are accepted.

: ACClass-AddChar   \ ch --
\ append the given character to the current window text.
    .....
;
: ACClass-WM_CHAR    \ --
\ WM_CHAR message handler
    wParam 32 127 within? if wParam ACClass-AddChar then
;
WinClass: ACClass
    Name "GUIgen-XXX-ACClass"
    CS_HREDRAW CS_VREDRAW or    Style
    CS_DBLCLKS                  Style+
    Color_Window Brush-colour

    WM_CHAR     does ACClass-WM_CHAR
End-WinClass

ACClass AppControl: TextEntry
\ make a control called TextEntry
    WS_CHILD WS_BORDER or       Style
    WS_HSCROLL WS_VSCROLL or    Style+
    WS_TABSTOP                  Style+
End-Control

Another TextEntry DigitEntry
\ make another TextEntry-type control which accepts only digits.
    WM_CHAR does: wparam 0 9 within not if -inherited then ;
end-control
```

4.9 Miscellaneous functions

```
: Enable-Control     \ ID --
```

Enables the control with a given ID which is a child of the current window.

```
: Disable-Control    \ ID --
```

Disables the control with a given ID which is a child of the current window.

```
: ControlEnabled    \ flag ID --
```

If flag is true, the control in the current window which has the given ID is enabled, otherwise it is disabled.

```
: Control-Enabled?   \ ID -- flag
```

Returns TRUE if the control with a given ID is enabled, otherwise FALSE is returned. The control must be a child of the current window.

```
: Hide-Control      \ ID --
```

Hides the control with given ID which is a child of the current window.

```
: Reveal-Control    \ ID --
```

Reveals (un-Hides) the control with given ID which is a child of the current window. (A synonym of SHOW-CONTROL)

```
: ControlShown  \ flag ID --
```

If flag is true, the control in the current window which has the given ID is show (made visible) otherwise it is enabled.

```
: Show-Control      \ ID --
```

Shows the control with given ID which is a child of the current window. (A synonym of REVEAL-CONTROL)

```
: Set-ControlTextz  \ z$ ID --
```

The child control of the current window whose ID is gven is set to the text string which lies at the address z$. The string must be a zero terminated string.

```
: Set-ControlText   \ add len ID --
```

Sets the control whose ID is given and is a child of the current window to the text specified by 'add len'.

```
: Get-CtrlText      \ add len ID -- add len'
```

Copy at most len characters of the controls text to the buffer at the given address. Return the start address and number of character copied. The copied text includes a terminating null character, which is not included in the returned length.

```
: Paint-Control     \ ID --
```

Causes the child control of the current window to be painted. Calls the API function InvalidateRect.

```
: Repaint-Control
```

Causes the child control of the current window to be painted, with the background erased first. Calls the API function InvalidateRect.

The following words provide a way of controlling which window

```
: NextTabCtl    \ hCtl1 -- hCtl2|0
```

Return the handle of the control which is next in the tab order to the control given. If no such control exists the value returned is 0.

```
: PrevTabCtl    \ hCtl1 -- hCtl2|0
```

Return the handle of the control which is before the control whose handle is hCtl1 in the tab order to the control given. If no such control exists the value returned is 0.

```
: FirstTabCtl   \ hDlg -- hCtl
```

Return the handle of the first 'tabbable' control which is a child of the window (usually a dialog box) with handle hDlg.

```
: LastTabCtl    \ hDlg -- hCtl
```

Return the handle of the first 'tabbable' control which is a child of the window (usually a dialog box) with handle hDlg/

```
: TabTo     \ hCtl -- flag;
```

'Tab' to the control whose handle is hCtl - that is the input focus is set to that control. The value returned is TRUE on success and FALSE on failure. If the input parameter is zero, no action is taken and the return value is FALSE;

4.10 Returning/Reflecting messages ...

Child controls often send 'notification' messages to their parent window which carries out an appropriate action in response. This can violate the object oriented doctrine of objects performing their own processing.

Sometimes a control should handle its own notification messages. To allow this a parent should return WM_NOTIFY or WM_COMMAND messages. Here is a way to do this.

The routines below are intended for use within a parent winproc routine (i.e. a dialog's DlgProc or controls CtlProc), and assume that Currxxx user variables contain the WinProc parameters.

`: Forward-Message \ --`
Forward the message to the current window's parent using the Windows function SendMessage, and storing any returned value in RETURNVAL. Used to forward WM_NOTIFY and WM_COMMAND messages.

4.11 Windows Defined Controls

`WS_CHILD WS_VISIBLE or WS_TABSTOP or WS_GROUP or constant WS_CONTROL`
A constant value which is the basis for most control's window style.

4.11.1 Buttons

Standard Buttons

`WinControl: PushButton "button"`
A standard push button control.

`WinControl: DefPushButton "button"`
A standard push button control but with the BS_DEFPUSHBUTTON style.

Check Boxes

A checkbox is a square box whith a text label by its side. A check box can be in a 'checked state' (the square box contains a cross or a tick), or in an 'unchecked state' (the square box is empty). A third state is possible for the so-called '3State' check boxes when the box is shown gray. Though check boxes don't look like buttons they are based on the Windows button class.

There are four types of check box:

`WinControl: CheckBox "button"`
A standard check box. The application is responsible for setting its state (to checked or unchecked) - usually by the parent window in response to a BN_CLICKED message sent by this control.

`WinControl: 3stateBox "button"`
A three-state check box - checked, unchecked or indeterminate. This control sends BN_CLICKED messages to its parent which will normally set the appropriate state of this control.

`WinControl: AutoCheckBox "button"`
A check box which sets its own state - each mouse click toggles its current state.

`WinControl: Auto3State "button"`
A three-state check box which sets its own state.

Radio Buttons

A Radio Button is a text label by the side of a small circle which is coloured when in the checked state. There are two types of radio button : A standard radio button relies on its parent to set its state to checked or unchecked. An AutoRadioButton toggles its own state and in addition ensures all other radiobutttons (of either type) in its group have the other state.

Note that radio buttons are created WITHOUT the WS_TABSTOP style even if that style was specified when created. When a radiobutton is checked it gains the WS_TABSTOP style, when it is put into the unchecked state its WS_TABSTOP style is removed.

`WinControl: RadioButton "button"`
A standard radio button. The application is responsible for setting its state.

`WinControl: AutoRadioButton "button"`
An AutoRadioButton which sets its own state and that of all other radio buttons in its group.

Button 'helper' words

`: CheckButton  \ hCtl --`
Check/Set the button with the given handle.Typically used in response to a WM_COMMAND message if the button is not an AUTO... button.

`: UnCheckButton \ hCtl --`
UnCheck/UnSet the button with the given handle.Typically used in response to a WM_COMMAND message if the button is not an AUTO... button.

`: SetButtonCheck  \ flag hCtl --`
Set the check state of button with the handle *hCtl* to *Checked* if flag is true, otherwise set it to *Unchecked*.

`: ButtonChecked?  \ hCtl -- flag`
Return true if the button with the handle hCtl is checked.

The Group Box

The group box is a peculiar type of button : it has no check state, focus state, or push state and cannot be selected. An application cannot send messages to a group box.

A group box consists of a rectangular frame and a text label. Its sole purpose is to provide a visual organisation of other controls.

`WinControl: GroupBox "button"`
A Group Box control.

4.11.2 Edit controls

An edit control is a rectangular window which allows text to be entered and edited from the keyboard. Two basic styles are available: single-line edit controls and multi-line edit controls. In addition, a number of edit control styles determine how text is formatted and shown.

The following styles are defined for edit controls :

• ES_AUTOHSCROLL and ES_AUTOVSCROLL controls scrolling.
• ES_CENTER, ES_LEFT, ES_RIGHT aligns text within the control.
• ES_LOWERCASE and ES_UPPERCASE forces lowercase or uppercase.
• ES_MULTILINE Designates a multiline edit control.
• ES_NOHIDESEL retains highlighted text even without focus.
• ES_NUMBER Allows only digits to be entered into the edit control.
• ES_OEMCONVERT Converts text to the OEM character set.
• ES_PASSWORD Displays an asterisk (*) for each character.
• ES_READONLY Prevents the typing or editing of text
• ES_WANTRETURN Allows carriage return be inserted in multi-line edit controls.

A small number of edit controls are defined below. Use the above styles and the WS_xxx window styles to create other variations using `Another`.

`WinControl: EditBox "edit"`
A standard single line edit control. This control is commonly referred to as EDITTEXT in Windows resource files. (I think EditText soundes too much like a verb, rather than nown.)

`WinControl: ViewBox "edit"`
A single line 'read-only' edit control.

`WinControl: Multi-EditBox "edit"`
A multi-line edit control. The WM_KEYDOWN message is trapped in this control and if a *Tab* key is detected an attempt to set the input focus to the next (or previous) control.

Operations on edit controls....

In the following glossary, the symbol 'hCtl' indicates the Windows handle of an edit control.

```
: Edt-SetMaxChars   \ n hCtl -- ;
```
Set max number of characters for and edit control to n.

```
: Edt-MaxChars      \ hCtl -- n
```
Get the maximum number of characters for an edit control.

```
: Edt-Changed?      \ hCtl - flag
```
Returns TRUE if the contents of the edit control has changed since it was created or since the 'changed flag' was set to FALSE. (see EDT-HASCHANGED and EDT-UNCHANGED)

```
: Edt-HasChanged    \ hCtl -- ;
```
Set the edit control's internal 'changed flag' to TRUE.

```
: Edt-UnChanged     \ hCtl -- ;
```
Set the edit control's internal 'changed flag' to FALSE.

```
: Edt-Clear         \ hCtl -- ;
```
Clear the contents of the edit control and mark it as unchanged.
Windows refers to text selected in edit controls using the 'positions' of first and last characters. The following words relating to selected text (Edt-Select and Edt-Selected) use the more familiar Forth convention of 'address length' values, with the address of the first character being zero.

```
: Edt-Select        \ start len hCtl --
```
Mark some text within the edit control as 'selected'. start is the starting position of the first character selected. len is the number of characters selected.

```
: Edt-Selected      \ hCtl -- start len
```
Return the starting position and length of the selected text in the edit control

```
: Edt-ReplaceText   \ add len hCtl --
```
Replace the text currently selected within an edit control by the text string specified by 'add len'.

```
: Edt-Appendz       \ z$ hCtl --
```
Append the zero-terminated string z$ to the given edit control.

```
: Edt-Append        \ add len hCtl --
```
Append the text string 'add len' to the text in the edit control.

```
: Edt-AppendLine    \ add len hCtl --
```
Adds the string given by 'add len' as a new line. This is acheived by adding a CR/LF character pair to the control and then appending the string.

```
: Edt-LineLen       \ hCtl line# -- len
```
Returns the length of line number 'line#' of the edit control.

```
: (Edt-GetLine)     \ hCtl line# buff len -- buff len'
```
Using the EM_GETLINE message, copies the text of line number line# to the given buffer. The size of the buffer is 'len' characters (bytes). Returned is the buffer address and the number of characters copied.

```
: Edt-GetLine       \ hCtl line# -- add len
```
Return the text of line number 'line#' at address 'add' and length 'len'. Note that the returned addres is in the Forth dictionary and may not be 'thread safe'.

```
: Edt-LineCount     \ hCtl -- n ;
```
Return the number of lines as reported by Windows. (BUT... see the word EDT-#LINES below.

```
: Edt-#Lines        \ hCtl -- n ;
```
Return the TRUE number of lines in an edit box. Windows can/does include any trailing blank lines in the count, this word removes it.

4.11.3 Static Controls

Static controls are controls which allow applications to show text or graphics that require no user input.

Perhaps the most common static controls are the text static controls sometimes known as 'labels', which simply display text. Others display frame outlines, coloured rectangles and images.

Regarding window styles, a static control will not normally have the WS_TABSTOP style. Don't do it - you don't want to move to a control which cannot take 'the focus'.

Originally, Win16 static controls were so called because they could not respond to mouse clicks and keyboard input. This has been changed to allow static controls to respond to mouse clicks; the term "static" being made misleading.

There are many static control styles, the SS_xxx values. Many of these styles cannot be mixed (ORed). For instance it took me some time I remember, to realise that a text style cannot be combined with a frame style. These SS_xxx values are NOT bit flags. The good thing is that if you want a frame control to also show text, a special control is relatively easily built.

`WinControl: LeftText "static"`
A simple control to display text. The background to the text seems to be COLOR_BTNFACE. See `TextLabel` later if you want text printed on a transparent background. This control prints text on a single line truncating text extending beyond its right hand side.

`WinControl: TextBox "static"`
Similar to the LEFTTEXT control above but will wrap text onto multiple lines.

`WinControl: CTEXT "static"`
A TextBox type control with text aligned centrally.

`WinControl: LTEXT "static"`
A TextBox type control.

`WinControl: RTEXT "static"`
A TextBox type control with text displayed right aligned.

`WinControl: EtchedFrame "static"`
A simple 'frame' type control which can be used to surround other controls.

`WinControl: WhiteFrame "static"`
A white coloured frame/box. This control has the SS_WHITEFRAME style. Black or gray frames can be created by creating similar controls and specifying SS_BLACKFRAME or SS_GRAYFRAME styles.

`WinControl: IconHolder "static"`
A control which is able to display icon images. To show an icon image see SET-ICON below.

```
: Set-Icon  \ hIcon hCtl --
```
Given a Windows handle to an icon image, 'set it' into the static control with handle hCtl. The control must have the SS_ICON style such as ICONHOLDER above.

```
: Get-Icon  \ hCtl -- hIcon|null
```
Given the handle to a static control, returns the handle of its icon, or null if it has no icon.

`WinControl: BitmapHolder "static"`
A control which will display a bitmap. Use `Set-BitMap` (defined below) to display a bitmap in this control.

```
: Set-BitMap    \ hBitMap hCtl --
```
Associates a bitmap to static control which must have the SS_BITMAP style (such as a BITMAPHOLDER). hBitMap is the Windows handle of the bitmap.

```
: Get-Bitmap    \ hCtl -- hBitMap
```
Returns the handle to the bitmap associated with the given static control. Returns NULL if none exists.

4.12 Enhancing Window Controls 1 - A transparent label

To provide a text control which shows its text without 'blocking out' the window beneath, the TextLabel control shown below was created.

Three window messages are 'intercepted' : WM_PAINT, WM_ERASEBKGND and WM_SETTEXT. The new paint action draws the control's text on a transparent background, and formats it according to the SS_LEFT, SS_RIGHT, SS_CENTER styles. The text will be printed on multiple lines unless it has the SS_SIMPLE style.

Painting the control uses the Windows API function DrawText, which requires the address and length of the text to draw, a handle to a device context on which to draw it, and a RECT structure defining a bounding rectangle. In addition is required a set of bit flags which control how the text is formatted and which are determined by inspecting the SS_xxx style values of the control.

The WM_ERASEBKGND action is a 'do nothing'. But windows has to be told to do nothing also.

The third message handling routine is for the WM_SETTEXT message. This was found neccessary to force the new text to be displayed. Since the control's text is printed on a transparent background, a rectangle of the parent window is invalidated, the size and position of the control.

Finally two controls are created : TEXTLABEL the transparent text control, and BORDERED-LABEL which is similar but shows a border surrounding the text.

To hide posible name conflicts, the words are placed in its own vocabulary.

```
\ vocabulary TextLabels
\ also TextLabels definitions

: TxtFmt    \ -- u
\ Returns the text formatting value for the control.
    CurrWin GWL_STYLE GetWindowLong
    dup SS_RIGHT and    if  DT_RIGHT    else
    dup SS_center and   if  DT_center   else
                            DT_LEFT
                        then then

    swap SS_SIMPLE or SS_SIMPLE =
    if DT_SINGLELINE else DT_WORDBREAK then
    or
;

: tl-PAINT      \ --
\ The text label WM_PAINT handler. Draws the controls text on a
\ transparent background.
                    { | R[ RECT ] txt[ 1 kB ] -- }

    CurrWin R[ GetClientRect drop       \ -- ; get window size
    CurrWin txt[ 1 kB 1- GetWindowText  \ len ; get text
    TxtFmt                      \ len fmt ; text format flags

    Start-Painting
    CurrDC TRANSPARENT SetBkMode drop \ len fmt
    CurrDC txt[ 2swap           \ hDC txt len fmt
    R[ swap                     \ hDC txt len rect fmt
    DrawText drop               \ -- ; all done
    Finish-Painting             \ --
;
```

```
: tl-ERASE      \ --
\ The WM_ERASEBKGND action
    1 ReturnVal !
    -WinHandler
;

: tl-SetText    \ -- ; WM_SETTEXT action
                { | R[ RECT ] -- }
    inherited
    CurrWin R[ GetClientRect drop   \ get area to be redrawn
    CurrWin dup GetParent
    R[ 2 MapWindowPoints drop       \ convert to parent coordinates
    CurrWin GetParent R[ 1
    InvalidateRect drop             \ and tell parent..
;

WinControl: TextLabel "static"
    WS_CHILD WS_VISIBLE or Style
    WS_EX_TRANSPARENT   ExStyle

    WM_PAINT            does tl-Paint
    WM_ERASEBKGND       does tl-Erase
    WM_SETTEXT          does tl-SetText
end-control

Another TextLabel BorderedLabel
    WS_Border Style+
end-control
```

5 Miscellaneous window functions.

This code provides a number of generally useful words for manipulation and managing windows. Often the word is simply an API call with 'hard coded' parameters.

5.1 Glossary

```
: Paint-Window      \ hWnd --
```
Paint the given window by calling the API function InvalidateRect. The window will not be erased first.

```
: Repaint-Window    \ hWnd --
```
Paint the given window by calling the API function InvalidateRect. The window WILL be erased first.

```
: RedrawWin \ hwnd --
```
Another way to cause the client area of a window to be re-painted. Try this if PAINT-WINDOW fails.

```
: Enable-Window     \ hWnd --
```
Enable the given window.

```
: Disable-Window    \ hWnd --
```
Disable the given window.

```
: CloseWindow   \ hWnd -- ; send a close message
```
Send a WM_CLOSE message the the window with handle hWnd

```
: @hInstance     \ -- u
```
Return the current instance handle.

```
: CreateDisplayDC   \ -- hdc | 0
```
Create a device context for the DISPLAY device. The returned value hDC is a handle to the device context which should be deleted using DeleteDC when it is finished with.

Concerning colours ...

```
: RGB   \ red green blue -- ColorRef
```
Converts the three (byte) values red green and blue into a 24-bit value or COLORREF.

```
: Colour:   \ red green blue <name> -- ; (child) -- n
```
A 'colour' creating word taking the thre RGB balues and creating a (24-bit) named constant. The child word returns the 24-bit value. The following definitions provide "standard" colours.

```
$00 $00 $00 Colour: Black
$00 $00 $FF Colour: Blue
$00 $FF $00 Colour: Green
$00 $FF $FF Colour: Cyan
$FF $00 $00 Colour: Red
$FF $00 $FF Colour: Magenta
$FF $FF $00 Colour: Yellow
$FF $FF $FF Colour: White
$C0 $C0 $C0 Colour: LtGrey
$80 $80 $80 Colour: Grey
$40 $40 $40 Colour: DkGrey
```

Handling rectangles etc.

Windows make much use of the its RECT structure - a record containing four fields labelled *rect.left*, *rect.right*, *rect.top* and *rect.bottom* in VFX - for various operations related to drawing in windows.

The related Windows POINT structure is also widely used. It consists of two fields point.X and point.Y, to hold the coordinates of a point in a window.

`: .Coords \ x y --`
print the x y coordinate in the format "(x, y)"

`: .Point \ 'point --`
print the X and Y values held in the POINT structure given.

`: .Rect \ rect --`
Print the top left and bottom right coordinate pairs of the given rectangle

`: .Dims \ x y w h --`
Print the coordinates (x,y) the width and height and the coordinate point (x+w, y+h).

`: Rect! \ x y r b rect --`
Stores the values x y r b in the RECT structure at the address given.

`: Rect@ \ rect -- l t r b`
Retreives the values x y r b from the RECT structure at the address given.

`: RectWidth \ rect -- w`
Find the width of the given rectangle.

`: RectHeight \ rect -- h`
Find the height of the given rectangle.

`: RectCentre \ rect -- x y`
Return the coordinates of the centre of the given rectangle.

`: ExpandDims \ x y w h n -- x' y' w' h'`
Expand the rectangular area defined by the given position (x and y) and size (w and h). x and y are reduced by n, while w and h are increased by 2n.

`: ShrinkDims \ x y w h n -- x' y' w' h'`
Reduce the size of the rectangular area defined by the given position (x and y) and size (w and h). x and y are increased by n, while w and h are decreased by 2n.

`: ExpandRect \ addr n --`
Expand the rectangular area defined by the values of the RECT structure at addr.

Window dimensions...

The following words are mostly concerned with either obtaining or setting a window property relating to its size and or position. The following notation is used in the stack comments:

- *hWnd* is a window handle.
- *w* and *h* represent a width and height.
- *x* represents a horizontal position in pixels.
- *y* represents a vertical position increasing downwards.

```
: WinWidth      \ hHdl  -- w
```
Return the width of the given window.

```
: WinHeight     \ hHdl  -- h
```
Return the height of the given window.

```
: WinSize       \ hWnd -- w h
```
Return the width and height of the given window.

```
: ScreenPos     \ hWnd -- x y
```
Given a window handle, return the screen coordinates of the top left hand corner of the window. Coordinates are given in pixels relative to the top left of the screen. You can convert to client coordinates using Screen>client. See also WinPosition.

```
: Screen>Client \ x y hWnd -- x' y'
```
A call to the window API function ClientToScreen but which avoids using a POINT structure. The values x and y represent the screen coordinates of a point. x' and y' are that same point expressed relative to the window client area whos handle is hWnd.

```
: Client>Screen \ x y hWnd -- x' y'
```
A call to the window API function ScreenToClient but which avoids *you* using a POINT structure. x' and y' represent the same point in screen coordinates.

```
: ParentWindow  \ hWnd1 -- hWind2
```
Return the handle of the parent window or the desktop window if there is no parent.

```
: Parent>Screen \ x y hWnd -- x' y'
```
Map the coordinates x and y relative to the window's parent client area, to screen coordinates x' and y'.

```
: Screen>Parent \ x y hWnd -- x' y'
```
Map the screen coordinates x and y to the window's parent client area, to give coordinates x' and y'.

```
: WinPos        \ hWnd -- left top ; hWnd -- x y
```
Return the coordinates of the top left corner of the given window. For child windows the returned values are relative to the window's parent client area. Non-child windows return values relative to the screen.

```
: WinDims       \ hWnd -- x y w h
```
Get the position and size of the window, relative to the windows parent client area.

```
: WinBottom     \ hWnd -- n
```
Returns the position of the bottom of the given window in its parent client coordinates.

```
: WinLeft       \ hWnd -- n
```
Returns the position of the left hand side of the given window in its parent client coordinates.

```
: WinRight      \ hWnd -- n
```
Returns the position of the right hand side of the given window in its parent client coordinates.

```
: WinTop        \ hWnd -- n
```
Returns the position of the top of the given window in its parent client coordinates.

```
: Set-WinDims   \ x y w h hWnd --
```
Set the position and size of a window relative to its parent's client area.

```
: Set-WinPos    \ x y hWnd --
```
Set the position of a window relative to its parent's client area.

```
: Set-WinWidth      \ w hWnd --
```
Set the width of a window.

```
: Set-WinHeight     \ h hWnd --
```
Set the height of a window.

```
: Set-WinLeft       \ x hWnd --
```
Set the left hand position of a window relative to its parents client area.

```
: Set-WinTop    \ y hWnd --
```
Set the top position of a window relative to its parents client area.

```
: Set-WinSize       \ w h hWnd --
```
Set the width and height of a window.

```
: Set-WinRight      \ x hWnd --
```
Position the window hWnd so that its right hand edge is at x. No change of window size occurs.

```
: Set-WinBottom     \ y hWnd --
```
Position the window hWnd so that its bottom edge is at y. No change of window size occurs.
The following synonyms are available but will be removed at some time in the future.

Synonyms

```
Alias: WinPosition WinPos               \ hWnd -- left top ; hWnd -- x y
Alias: WinDimensions WinDims            \ hWnd -- x y w h
Alias: Set-WinDimensions Set-WinDims    \ x y w h hWnd --
Alias: Set-WinPosition Set-WinPos       \ x y hWnd --
```

The following synonyms are now prefered becouse of their brevity. They will replace the original ones in the future...

Preferred word synonyms

```
Alias: WinDims!     Set-WinDims   \ x y w h hWnd --
Alias: WinPos!      Set-WinPos    \ x y hWnd --
Alias: WinWidth!    Set-WinWidth      \ w hWnd --
Alias: WinHeight!   Set-WinHeight     \ h hWnd --
Alias: WinLeft!     Set-WinLeft       \ x hWnd --
Alias: WinTop!      Set-WinTop    \ y hWnd --
Alias: WinSize!     Set-WinSize       \ w h hWnd --
Alias: WinRight!    Set-WinRight      \ x hWnd --
Alias: WinBottom!   Set-WinBottom     \ y hWnd --
```

```
: ClientSize    \ hWnd -- w h
```
Returns the width and height of the client area of the window whose handle is hWind.

```
: ClientWidth   \ hWnd -- w
```
Returns the width of the client area of the window whose handle is hWind.

```
: ClientHeight  \ hWnd -- h
```
Returns the height of the client area of the window whose handle is hWind.

```
: Set-ClientSize    \ w h hWnd --
```
Sets the width and height of the client area of the window whose handle is hWind.

`: AtCentreOf  \ hWnd1 hWnd2 --`
Position window hWnd1 at the centre of hWnd2. If hWnd is a child window it must be a child window of hWnd2

Concerning window text.

The following words retrieve and set window text with various text formats. Windows usually uses zero-terminated strings but occasionally deals with address and length. What is meant by "window text" depends on the window. Normally it refers to the caption of a window. For an edit control however it is the text being edited.

`: get-WinText  \ hWnd addr ulen -- addr ulen'`
Copies up to *ulen* characters of text from the window with handle *hWnd* to memory starting at the address *addr*. *ulen'* is the number of characters copied. This is a wrapper around the WM_GETTEXT message.

`: WinText  \ hWnd -- add len`
Return (up to 1 Kb of) text from the window with the given handle. The returned text is contained in a static buffer, which will be re-used by this word.

`: WinTextz  \ hWnd -- caddrz`
Return (up to 1 Kb of) text from the window with the given handle. The returned text is contained in a static buffer, which will be re-used by this word.

`: WinTextLen  \ hWnd -- len`
Returns the length of the window text.

`: WinText?  \ hWnd -- flag`
Returns the TRUE iff the length of the window text is non-zero.

`: Set-WinTextZ  \ textz hWnd --`
Sets the text for the window with handle *hWnd* to the text specified as the address of a zero-terminated string.

`: Set-WinText  \ add len hWnd --`
Sets the text for the window with handle *hWnd* to the text specified as an address and length pair.

`: BeginUpdate  \ hWnd -- ; inhibit changes causing a redraw`
Execute this word to prevent changes in that window from being redrawn. Use `EndUpdate` to allow changes to be redrawn. Use this before adding lots of items to a list box.

`: EndUpdate  \ hWnd -- ; allow a change to cause a redraw`
Execute this word after `BeginUpdate` to allow to changes to be redrawn.

`: IsWindow?  \ hWnd -- flag`
Returns a well formed flag which is TRUE only if hWnd is the handle to a window.

Simple Sounds

`: Exclaim  \ --`
Does the API call MessageBeep with the MB_ICONEXCLAMATION parameter.

`: Blip  \ --`
Does the API call MessageBeep to make a standard beep;

`: WinNT?  \ -- flag`
Returns TRUE if the OS is Windows NT

```
: Win95?    \ -- flag
```
Returns TRUE if the OS is Windows 95 or Windows 98

```
: OSmajor   \ -- u
```
Return the major version number of the OS.

```
: OSminor   \ -- u
```
Return the minor version number of the OS.

Window Styles

The following words manipulate the style of a window by using the two API functions SetWindowLong and GetWindowLong. Note that this may not allways change the windows appearance some styles seem to have an effect only if created with them.

```
: WinStyle   \ hWnd -- style
```
Get the given window's style.

```
: Set-WinStyle   \ style hWnd --
```
Set the given window's style.

```
: WinStyle+      \ style hWnd --
```
Add the given style to the window.

```
: WinStyle-      \ style hWnd --
```
Remove the given style from the window.

```
: WinExStyle     \ hWnd -- style
```
Get the given window's extended style.

```
: Set-WinExStyle     \ ExStyle hWnd --
```
Set the given window's extended style.

```
: WinExStyle+    \ ExStyle hWnd --
```
Add the extended style bits to the specified window.

```
: WinExStyle-    \ ExStyle hWnd --
```
Removed the extended style bits from the specified window.

Finding children and sibling windows.

```
: GetWindow:    \ n "<name>" ++ ; (child) hWnd1 -- hWnd2
```
A defining word whose child words call the Windows API function GetWindow. See below for the words defined by this one.

```
GW_CHILD        GetWindow: TopWindow    \ hWnd1 -- hWnd2
```
Returns the child window which is the top-most window (in the z-order) of the given parent.

```
GW_HWNDFIRST    GetWindow: FirstWindow  \ hWnd1 -- hWnd2
```
Returns the sibling window which is highest in the z order. If hWnd1 identifies a Topmost window or a top-level window, hWnd2 is a a Topmost window or a top-level window.

```
GW_HWNDLAST     GetWindow: LastWindow   \ hWnd1 -- hWnd2
```
Returns the sibling window which is lowest in the z order.

```
GW_HWNDNEXT     GetWindow: NextWindow   \ hWnd1 -- hWnd2
```
Returns the sibling window next lowest in the z order.

```
GW_HWNDPREV     GetWindow: PrevWindow   \ hWnd1 -- hWnd2
```
Returns the sibling window next highest in the z order.

```
GW_OWNER        GetWindow: OwnerWindow  \ hWnd1 -- hWnd2
```
Returns a handle to the given window's owner, if any.

Finding and Setting z-orders

```
: MakeTopWindow \ hWnd --
```
Make the window whose handle is hWnd the top-most window. This is a call to the API function BringWindowToTop.

```
: Z-Order   \ hWnd -- pos ;
```
Return the position of the window whose handle is hWnd, in the z-order within the parent window.

```
: WinTopper    \ u "name" ++
```
A defining word whose children set the window positioning flags of a window to u.

```
HWND_BOTTOM     WinTopper   TopWin-          \ hWnd --
```
Places a window at the bottom of the z-order.

```
HWND_NOTOPMOST  WinTopper   TopMostWin-      \ hWnd --
```
Places a window above all non-topmost windows and behind all topmost windows.

```
HWND_TOP        WinTopper   TopWin+          \ hWnd --
```
Places a window at the top of the z-order.

```
HWND_TOPMOST    WinTopper   TopMostWin+      \ hWnd --
```
Places a window above all non-topmost windows.

```
: PutBehind     \ hWnd1 hWnd2 --
```
Place window hWnd1 behind hWnd2 on the screen.

```
: SendToBack    \ hWnd --
```
Makes a window the the lastmost window

```
: BringToFront  \ hWnd --
```
Brings the window to the front of all it siblings.

```
: .Children     \ hWnd --
```
Lists the handles of all the given windows children.

Miscellaneous Functions

```
: CountChildren     \ hWnd -- n
```
Uses the Windows API procedure *EnumChildWinProc* to count the number of children a window has.

```
: AllChildren   \ xt hWnd --
```
Execute the word whose xt is given against all child windows of the window whose handle is hWnd. The stack effect of the word is (hWnd –).

```
: DisableChildren   \ hWnd --
```
Disables all child window of the given window.

```
: EnableChildren    \ hWnd --
```
Enables all child window of the given window.

```
: Hide-Window    \ hWnd --
```
Hide the window.whose handle is hWnd.

```
: Reveal-Window  \ hWnd --
```
Show a previously hidden window.

`: Minimise-Window \ hWnd --`
Minimise a window.

`: Maximise-Window \ hWnd --`
Maximise a window.

`: Restore-Window \ hWnd --`
Restore a window

`: Normal-Window \ hWnd --`
Show the window restoring it to its original size and position.

`: DrawWindowFrame \ hWnd -- n`
Redraws a window frame.

`: wVisible? \ hWnd -- flag`
Return true if the window is visible (has the WS_VISIBLE window style.

`: CentreWin \ hWnd --`
Position the window at the centre of its parent window. If the window does not have a parent window its owner window is used. If there is no owner window it is positioned at the centre of the desktop window.

Fonts

`: WinFont \ hWnd -- hFont`
Return the handle to the window by sending a WM_GETFONT message.

`: Set-WinFont \ hFont hWnd --`
Sends a WM_SETFONT message to set a windows font.

`: Set-Children'sFont \ hFont hWnd --`
hWnd is the handle of a window whose children each has their font set to the font with handle hFont.

5.2 Keyboard state checks

The following functions are 'wrappers' around the Windows API function

SHORT GetKeyState(int *nVirtkey***)**

which takes the ASCII value of a character or a virtual key code and returns the 'status' of the corresponding key:

- The key is down iff the high-order bit is set.
- The key is 'toggled' iff the low order bit is set. (Toggled here means 'turned on' such as the `CAPS LOCKED` key).

`: Key-Pressed? \ keycode -- flag`
Returns TRUE iff the key corresponding to the given key code is down.

`: Key-On? \ keycode -- flag`
Returns TRUE iff the key corresponding to the given key code is 'toggled'.

`: PressTest: \ keycode "<name>" ++ ; Child: -- flag`
A defining word. Children return TRUE iff the key corresponding to the given key code is down.

5.2.1 NOTE: This code calls the API function GetKeyState which

```
VK_CONTROL  PressTest:  CTL-Pressed?    \ -- flag
```
Returns TRUE if the Ctl key is depressed. Otherwise returns FALSE.

```
VK_Shift    PressTest:  Shift-Pressed?  \ -- flag
```
Returns TRUE if the Shift key is depressed. Otherwise returns FALSE.

```
VK_Menu     PressTest:  Alt-Pressed?    \ -- flag
```
Returns TRUE if the Alt key is depressed. Otherwise returns FALSE.

```
: VirtualScreen     \ -- x y w h
```
Return the position and size of the virtual screen. (The virtual screen is the bounding rectangle of all display monitors. The virtual screen is the bounding rectangle of all display monitors.

```
: ConfineToDesktop  \ hWnd --
```
Confine the window to the desktop.

```
: WorkArea  \ -- x y w h
```
Returns the size of the "working area" which is defined as the portion of the screen not obscured by the tray.

6 Menus

6.1 Introduction

These words provide a menu resource-builder, creating the API V4 type of menu template: a MENUEX_TEMPLATE_HEADER structure followed by MENUEX_TEMPLATE_ITEM structures.

The point of using these Windows-defined structure is to reduce some of the effort of creating menus. By building a complete menu definition using MENUEX_TEMPLATE_xxxx structures, a menu is easily created by calling the API function `LoadMenuIndirect`. This works well for a standard menu, one that Windows places along the top of a window. The method does not work for popup menus however - those 'short-cut' windows often activated by clicking the right hand mouse button. To create a popup menu an empty menu is created and the individual items are added to it. This makes using the MENUEX_TEMPLATE_xxx structures almost pointless but I stick with it in the hope that the missing function may appear.

In addition to the menu creation words other more general menu manipulation words are provided - words to disable and enable individual menu items etc.

An effort has been made to create a notation that resembles Windows resource scripts. The examples below illustrate how to define a main menu and a popup window. See the chapters about creating windows, dialogs and control for information on how to use menus with those various items.

The main menu definition below may be used in a window class definition and a dialog box definition.

```
NextID: IDM_Item10
NextID: IDM_Item11
NextID: IDM_Item12
NextID: IDM_Item13

NextID: IDM_Item20
NextID: IDM_Item21
NextID: IDM_Item22
NextID: IDM_Item23

Menu: Menu1
    IDM_Item10 SubMenu "&File"
        IDM_Item11 Item "&New"
        IDM_Item12 Item "&Open"
        IDM_Item13 Item "E&xit"
    end-menu

    IDM_Item20 SubMenu "&Edit"
        IDM_Item21 item "Cu&t"          greyed
        IDM_Item22 item "&Copy"         greyed
        IDM_Item23 item "&Paste"        greyed
    end-menu
end-menu
```

The next illustration is of a *popup menu* definition. A popup may be used within window, dialog box and control definitions, in which case the popup will appear when the right mouse button is clicked over that window.

```
NextID: IDM_Popup10
NextID: IDM_Popup20
NextID: IDM_Popup21
NextID: IDM_Popup22
NextID: IDM_Popup23

Popup: Popup1
    IDM_Popup10 item "&Delete"

    IDM_Popup20 SubMenu "&Clipboard"
        IDM_Popup21 item "Cu&t"          greyed
        IDM_Popup22 item "&Copy"         greyed
        IDM_Popup23 item "&Paste"        greyed
    end-menu
end-menu
```

6.2 Setting up

6.2.1 Vocabularies

Most of the menu code is compiled into a vocabulary called `Menus` thus 'hiding' it from casual

view. Words which are used to define menus are held in the vocabulary `DefiningMenus` and this is added to the search order by both `Menu:` and `Popup:`, and removed by `end-menu`.

`vocabulary menus`
The vocabulary into which most menu words are compiled.

`vocabulary DefiningMenus`
This vocabulary holds the menu definition words.

6.2.2 Windows structures

The Windows API function `LoadMenuIndirect` is used to create menus using the Windows *extended menu template* structures defined below. These structures are built at compile-time by the window defining words `Menu:` and those words in the `DefiningMenus` vocabulary.

`struct MENUEX_TEMPLATE_HEADER`
The extended menu template header structure. Words created by `Menu:` and `Popup:` build this structure.

`struct MENUEX_TEMPLATE_ITEM`
The extended menu template item structure. These structures are appended to children of `Menu:` and `Popup:` as menu items are added.

`: SubMenu?  \ addr -- flag`
addr is the address of a MENUEX_TEMPLATE_ITEM structure and *flag* is true if the item has been marked as being the start of a submenu.

`: EndMenu?  \ addr -- flag`
Returns a value of TRUE if the MENUEX_TEMPLATE_ITEM structure at the given address is marked as being the last item of a menu or submenu.

Wide character/'unicode' string operations

The API function **LoadMenuIndirect** requires the use of 16-bit Unicode strings. The following words provide help in manipulation of them. Thes words are compiled into the `Menus` vocabulary and are not intended for general use.

`: wzStrLen      \ addr -- n`
Returns the length of the zero-terminated wide-character (i.e. 16-bit character length) string. The length n does NOT include the terminating null.

`: wType     \ addr n --`
Wide character version of `TYPE`. n is the number of 16-bit characters.

`: wz$.      \ addr --`
Types the zero-terminated wide-character string.

Traversing the structures

Popup menus are created using the API function **CreatePopupMenu** at run-time and then adding individual menu items encountered while traversing MENUEX_TEMPLATE_xxx structures (compiled into the dictionary by `Popup:`). The following words help in that traversal.

`: SkipHdr          \ menu -- item[0]`
Given the base address of a menu, skip over the MENUEX_TEMPLATE_HEADER returning the address of the first menu item.

`: SkipText  \ addr1 -- addr2`

Given the address of a zero-terminated wide-character string, return the address immediately following. Note that the address is aligned on a 32-bit boundary in keeping with the MENUEX_TEMPLATE_ITEM specification.

`: Skip-MTI \ addr1 -- addr2 ;`
Skip over a complete MENUEX_TEMPLATE_ITEM at address *addr1* to the start of the following MENUEX_TEMPLATE_ITEM structure at *addr2*.

`: mtitem.HelpID \ menuitem -- 'HelpID`
Returns the address of the helpID field in the given MENUEX_TEMPLATE_ITEM address. This field is present only if the menuitem describes a submenu.

`: .Menu \ menu --`
For debugging/testing. Prints the menu definition to the console.

6.3 Current menu stack

The current menu stack, is used to manage submenus - menus within menus. The top stack item is used to store the address of the menu item currently being defined, and words within the DEFININGMENUS vocabulary use this to modify or add information to the item's fields. The word MENU: creates a named menu structure in the dictionary, and the menu stack is initialised/reset. Each SUBMENU pushes another value onto the stack, and END-MENU pops a value from the stack, restoring the previous 'current menu item'.

`16 cells buffer: MenuItemStack`
The menu item stack. It will take up to 16 items indicating that menus/submenus may not be more than 15 levels deep.

`MenuItemStack value CurrMenuItem \ -- addr`
A value which holds the address within the `MenuItemStack` of the "top" item. This is the current menu item. The stack is empty when this value is the address of `MenuItemStack`. Adding an item to the stack involves increasing `CurrMenuItem` to the next cell address.

`: MenuLevel \ -- n`
Returns the depth of "menu nesting" - that is how menu submenu-of-submenu levels there are.

`: ?CurrMenuItem \ --`
Aborts if there is no current menu.

`: New-Menu \ --`
Begins a new menu by increasing the `MenuLevel` by one and setting `CurrMenuItem` to zero.

`: First-Menu \ --`
Resets the `MenuItemStack` and sets `CurrMenuItem` to zero.

`: Prev-Menu \ --`
Decreases the `MenuLevel` value by dropping the top item on the `MenuItemStack`. Thus restoring the `CurrentMenuItem` to its value before the last `NewMenu`.

6.4 Creating the menu dictionary strucures

Helper words

`: "\t"? \ caddr u -- flag`
Menu text may contain a TAB character to help format menu item text. This words used returns true only if the the next two characters in the ASCII string given is the string `"\t"` (i.e. a slash followed by a letter t) which is used to indicate a TAB character.

`: MenuText,  \ caddr u --`
Lays down the given ASCII text as a wide character zero-terminated string at `HERE` in the dictionary. This forms part of the process of building a MENUEX_TEMPLATE_ITEM structure.

`: Build-MenuItem  \ ID caddr u --`
Builds a MENUEX_TEMPLATE_ITEM structure at `HERE`, filling it with the menu ID and text given.

`: Menu:  \ "<name>" ++`
The menu defining word. Creates a named dictionary item and builds a MENUEX_TEMPLATE_HEADER into the word's body. Initialised the menu item stack and adds `DefiningMenus` to the search order.

`Synonym Popup: Menu:`
The word used to define a popup menu. As of February 2006 this is now a synonym of `Menu`. The following words are used to define a menu once that `Menu:` and `Popup:` has been used.

`: End-Menu  \ --`
Finishes the current menu or submenu.

`: Item  \ ID "<text>" ++`
Creates a menu item by calling `Build-MenuItem`. ID becomes the menu item ID number and the menu item text is taken from the input stream.

`: SubMenu  \ ID "<text>" ++`
Starts a submenu and creates a menu item with the given ID and text taken from the input stream.

`: Separator \ ID --`
Creates a seperator in the current menu which has the ID given.

`: HelpID  \ u --  ;`
Sets the help ID field for the current menu item. Aborts if the current menu item is not a sub menu/popup menu.

`: SetsState`
A defining word to create words which modify the current menu item state field.

`MFS_CHECKED SetsState Checked`
Marks the current menu item as 'checked'.

`MFS_GRAYED  SetsState Greyed`
Marks the current menu item as 'greyed'.

`MFS_DEFAULT SetsState Default`
Marks the current menu item as 'default'.

`MFS_HILITE  SetsState Highlight`
Marks the current menu item as 'highlighted'.

6.5 Run-time menu manipulation

Manipulating individual menu item states

The words below which manipulate and modify menus are based on the API calls `SetMenuItemInfo` and `GetMenuItemInfo` along with the associated data structure `MENUITEMINFO` (already defined in VFX).

`MenuItemInfo Constant /MenuItemInfo`
A synonym of MenuItemInfo which is thought to be easier to read in Forth. Returns the size of a MenuItemInfo data structure.

`: ClearInfo \ 'MenuItemInfo --`
Given a MenuItemInfo structure, it is cleared and initialied ready for use. (The structure is `ERASE`d and its `MENUITEMINFO.cbSize` field set.

`: MIState@ \ ID hMenu -- state`
Retrieves the state of the menu item with the ID value given as *ID* from the menu whose handle is *hMenu*. `GetMenuItemInfo` is called and the value of the field `MENUITEMINFO.fState` is returned.

`: MIState! \ ID hMenu state --`
Sets the state of the menu item with the ID value given as *ID* from the menu whose handle is *hMenu* to the value *state*. Uses the API call `setMenuItemInfo`.

`: StateSetter: \ flags "<name>" ++ (compiling) ; ID hMenu -- (child)`
A defining word creating words to set the state bits of menu items. *flags* is the state bits to set and the menu item is defined by its ID *ID*, and its handle *hMenu*.

`: StateUnsetter: \ flags "<name>" ++ (compiling) ; ID hMenu -- (child)`
A defining word creating words to reset the state bits of menu items. *flags* is the state bits to set and the menu item is defined by its ID *ID*, and its handle *hMenu*.

`: StateChecker: \ flags "<name>" ++ (compiling) ; ID hMenu -- flag (child)`
A defining word creating words to test the state bits of menu items. *flags* is the state bits to set and the menu item is defined by its ID *ID*, and its handle *hMenu*.

`MFS_DISABLED StateSetter: mitem-Disable \ ID hMenu --`
Disable the specified menu item.

`MFS_HILITE StateSetter: mitem-HiLight \ ID hMenu --`
Highlight the given menu item.

`MFS_CHECKED StateSetter: mitem-Tick \ ID hMenu --`
Place a tick against the specified menu item.

`MFS_DEFAULT StateSetter: mitem-Default \ ID hMenu --`
Mark this menu item as the default. (Under Windows 98 at least this seems to simply make the item text bold).

`MFS_DISABLED StateUnsetter: mitem-Enable \ ID hMenu --`
Enable the menu item.

`MFS_HILITE StateUnSetter: mitem-LoLight \ ID hMenu --`
Removes the highlight from the given menu item.

NOTE: This does not work under Windows 98! The bit flag for the hightlight state cannot be removed programatically.

`MFS_CHECKED StateUnSetter: mitem-UnTick \ ID hMenu --`
Removes the tick from the specified menu item.

`MFS_DEFAULT StateUnSetter: mitem-UnDefault \ ID hMenu --`
Removes the "default" status form the menu item specified.

`MFS_CHECKED StateChecker: MItem-Ticked?`
Returns true if the menu item is marked as ticked.

`MFS_DISABLED  StateChecker:  MItem-Disabled?`
Returns true if the menu item is marked as disabled.

`MFS_HILITE  StateChecker:  MItem-HiLite?`
Returns true if the menu item is marked as highlighted.

`MFS_DEFAULT  StateChecker:  mitem-Default?`
Returns true if the menu item is marked as the default.

A different method of setting/resetting menu item properties

Miscellaneous menu manipulation

```
: #MenuItems    \ hMenu -- n
```
Returns the number of menu items belonging to the menu with handle *hMenu*.

```
: MItem>Pos     \ ID hMenu -- pos | -1
```
Given a menu handle and the ID of a menu item return its position within the menu. If that menu item cannot be found -1 is returned.

```
: MItem-Text    \ ID hMenu -- caddr ulen
```
Returns the text of a menu item which has been copied to a local buffer.

```
: MenuText  \ hMenu ID -- ca u
```
Returns the text of a menu item which has been copied to a local buffer.

```
: SetMenuText     \ ca u hMenu ID --
```
Set the text for the specified menu item.

7 Bitmaps

7.1 Introduction

This document contains much information about the file and menory structure of the various types of bitmap and program code is interspersed within that information. In practice very few words are commonly used, so for easy reference these are described below.

Summary Glossary.

```
: BitMap-Size       \ hBITMAP -- w h
```
Return the width and height of the bitmap whose handle is hBITMAP.

```
: BMF:  \ "name" "pathname" ++ ;
```
Creates a named word in the dectionary and copies the contents of the named file to its parameter field (or whatever it is called under ANSI). Pass that address to Load-BMF to get Windows to create a bitmap and return a bitmap handle. Remember to use DeleteObject when the handle has been finished with.

```
: Load-BMF      \ bmf -- hBM
```
Given the address of a bitmap file structure, get Windows to create a bitmap and return its handle.

```
: Load-BitmapFile   \ add len flags -- hBITMAP
```
Given the name of a bitmap file, create a bitmap and return its handle. Uses the Windows API funtion LoadImage. flags are a set of LR_xxx flags.

```
: Load-DIBitmap     \ add len -- hBITMAP
```
Given the file name of a bitmap file, load the file data as a device independant bitmap and return a handle to it. Uses the Windows API function LoadImage.

```
: Load-TransparentDIB   \ add len -- hBITMAP
```
Given the file name of a bitmap file, load the bitmap data and return a handle to it. Uses the Windows API function LoadImage. The colour of the first pixel of the image is deemed to be 'transparent' : all pixels of that colour are replaced with COLOR_WINDOW. This applies only to those images which have a corresponding colour tables.

```
: Icon:     \ "name" "pathname" ++
```
Create a dictionary word, and lay the contents of the given filename as its data area.

```
: Create-Icon       \ 'icon n -- hICON
```
Given the address of an icon resource structure create the n-th icon and return its Window handle. NOTE : the icons created have the "large icon" - which is usually 32 pixels square, even if the bits specify 16 pixels square.

```
: Load-IconFile     \ add len flags -- hICON
```
Create an icon and return its Windows handle, given its file name. Calls the Window API LoadImage in which 'flags' are the LR_xxx load flags.

```
: Load-Icon     \ add len -- hICON
```
Create an icon and return its Windows handle, given its file name. Uses Load-IconFile.

```
: Load-TransparentICON  \ add len -- hICON
```
Create a transparent icon wnd return its Windows given its file name. Uses Load-IconFile

```
: Std-IconSize      \ -- w h
```
Returns the size of a standard icon.

```
: small-Iconsize    \ -- w h
```
Returns the size of a small icon.

A simple example

To compile bitmap data into the dictionary simply use `BMF:`

```
BMF: TargetBMap %images%target.bmp
```

Before a bitmap can be used, Windows has to "read" it and give it a handle. Use `Load-BMF`:

```
TargetBMap Load-BMF
```

A button to show this bitmap can be created as follows:

```
Another PushButton BM-Button
    BS_BITMAP Style+
    :noname
        CurrWin BM_SETIMAGE IMAGE_BITMAP    \ hBtn msg type
        TargetBMap Load-BMF     \ hBtn msg type hBitMap
        SendMessage drop        \ --
    ; to-message GWM_InitControl
end-control
```

7.2 Device Independent Bitmaps

7.3 The OS/2-style DIB

A DIB file has four main sections :

- 1) A file header
- 2) An information header
- 3) An RGB colour table (but not always).
- 4) The bitmap pixel bits.

A DIB without the file header (of 14 bytes) is said to be in packed-DIB format.

The file header structure is defined as follows:

```
struct /BitmapFileHeader
    _word   field   bmfh.Type        \ signature:"BM" or $4d42
    _dword  field   bmfh.Size        \ entire size of file
    _word   field   bmfh.Reserved1   \ must be zero
    _word   field   bmfh.Reserved2   \ must be zero
    _dword  field   bmfh.OffsetBits  \ offset to pixel bits
end-struct
```

The second section - the core - means that this format is the basis (hence 'core') of other bitmap formats derived from it. It is defined as follows:

```
struct /BitmapCoreHeader
    _dword  field   bmch.Size       \ size of structure = 12
    _word   field   bmch.Width      \ width in pixels
    _word   field   bmch.Height     \ height in pixels
    _word   field   bmch.Planes     \ =1
    _word   field   bmch.BitCount   \ bits/pixel (1,4,8 or 24)
end-struct
```

The bmch.BitCount field is the number of bits per pixel. The number of colours in the image is a 'power of two' number. For a two-colour bitmap this number is 1 (one). It is 8 for a 256-colour DIB.

For 1, 4, and 8-bit DIB, the BitmapCoreHeader is followed by the colour table. The colour table does not exist for 24-bit DIBS.

The colour table is an array of 3-byte RGBTRIPLE structures, one for each colour of the image. The structure is defined as follows :

```
struct /RGBTrible
    _byte   field   rgbt.Blue
    _byte   field   rgbt.Green
    _byte   field   rgbt.Red
end-struct
```

It is recomended that a colour table be arranged such that the most important colours appear first.

Since a colour table (when present) has either 2, 16 or 256 3-byte entries, the data immediately following is on a word boundary.

7.3.1 The Pixel Bits

DIB Pixel bits are organised in horitontal rows (as are most bitmap formats), often know as 'scan lines' The DIB begins with the bottom row (that is the bottom row displayed). The reason is to do with co-ordinate systems - Y-axis increases upwards.

The number of lines is equal to the bmch.Height field, in the BITMAPCOREHEADER section. Each row encodes bmch.Width number of pixels, begining with the leftmost and proceding to the right.

The length of each row is ALWAYS a multiple of 4, calculated as

```
RowLength = 4*( bmch.Width *bmch.BitCount + 31)/32
```

For DIBs with 1 bit per pixel, the leftmost pixel is the most significant bit of the first byte:

```
Pixel   0  1  2  3  4  5  6  7    8  9 10 11 12 13 14 15 ...
Bit#    7  6  5  4  3  2  1  0    7  6  5  4  3  2  1  0 ...
```

A zero bit indicates a colour given by the first RGBTRIPLE in the colour table. A bit of one indicates a colour given by the 2nd RGBTRIPLE.

4-bit DIBS contain pixel data arranged as two pixels per byte as follows

```
Pixel   --- 0 ----  --- 1 ----    --- 2 ---- --- 3 ---- ...
        7  6  5  4  3  2  1  0    7  6  5  4  3  2  1  0 ...
```

Each 4-bit has a value of n in the range [0..15] indicating the n-th entry in the RGBTRIBLE colour table.

8-bit DIBs contain values 0 up to 255 - indexes into the 256 RGBTRIPLE values in the colour table.

```
Pixel   --------- 0 ----------    --------- 1 ---------- ...
        7  6  5  4  3  2  1  0    7  6  5  4  3  2  1  0 ...
```

24-bit DIBs are different. Each pixel requires 3 bytes for the red, green and blue colour values. Each row is an array of RGBTRIPLE structures, padded with '0' bytes to ensure it is a multiple of 4 bytes long. The blue is the first byte, followed by green then by red.

7.4 The Expanded Windows DIB

The Expanded DIB format was introduced to Windows at the same time as the OS/2 DIB structure. The Expanded DIB structure is similar in outline :

- 1) A file header
- 2) An information header
- 3) An RGB colour table (but not always).
- 4) The bitmap pixel bits.

The differences lie in sections 2 and 3.

7.4.1 The Expanded DIB Information Header

Section 2 of the Extended DIB is a BITMAPINFOHEADER structure rather than a BITMAP-COREHEADER structure. The BITMAPINFOHEADER structure is defined as follows:

```
struct /BitMapInfoHeader
    _dword  field   bmih.Size           \ = 40 (bytes)
    _long   field   bmih.Width          \ image width in pels
    _long   field   bmih.Height         \ height in pels
    _word   field   bmih.Planes         \ = 1
    _word   field   bmih.BitCount       \ 1,4,8,16,24 or 32
    _dword  field   bmih.Compression    \ compression code
    _dword  field   bmih.SizeImage      \ #bytes in image
    _long   field   bmih.XPelsPerMetre  \ horiz. resolution
    _long   field   bmih.YPelsPerMetre  \ vert. resolution
```

```
    _dword  field   bmih.ClrUsed        \ # colours used
    _dword  field   bmih.ClrImportant   \ # important colours
end-struct
```

The two DIB styles may be distinguished by inspecting the the bmih.Size field which has a value of 40 in an Extended DIB format, a value of 12 is OS/2 style. (But see the `/BitmapV4Header` structure below}.

```
: ExtendedDIB?      \ add -- flag
```

add is either a BitMapInfoHeader or a BitmapCoreHeader structure. A value of TRUE is returned if the value of its bmih.Size field is 40 which indicates a BitMapInfoHeader structure rather than the OS/2 style.
NOTE: The width and height fields are 32-bit values in this BitMapInfoHeader structure - those in the BitMapCoreHeader are 16-bits!

Some fields have been re-defined in Windows 95 and NT 4.0. For example the bmih.Height field may take a negative value which indicates a 'top-down' bit map, with origin at top left corner.

The bmih.XPelsPerMetre and bmih.YPelsPerMetre are not used by Windows but may be used by an application. Note that typical values are for a screen set to 72 dot per inch is about 2,835 pixels per metre. A common printer resolution is 300 DPI = 11.811 pixels per metre.

The bmih.ClrUsed field when used (non-zero) indicates the number of colour entries in a colour table. Thus a 8-bit image which uses fewer than 256 colours can save some space by using this field. For instance a value if 64 in this field implies that pixel values in the range [0..63] only are used.

With Windows 95 and NT 4.0, the BitCount field can contain values of 16 or 32 as well as the previous values of 1, 4, 8 and 24 . Also starting with Windows 95, the ClrUsed field can be non-zero for pixel sizes of 16, 24 and 32, and in these cases the colour table can/is used to set a palette for the EDIB.

The bmih.ClrImportant is usually set to zero. A non-zero value n say, means that the image can be rendered using only the first n entries in the colour table. A value of n other than zero or the value in the bmih.ClrUsed field is rare.

The Extended DIB (EDIB) structure has a colour table made up of RGBQUAD structures -

```
struct /RGBQuad
    _byte   field   rgb.Blue
    _byte   field   rgb.Green
    _byte   field   rgb.Red
    _byte   field   rgb.Reserved
end-struct
```

Another structure defined by Windows is the BITMAPINFO structure, which is just a BITMAP-INFOHEADER with a single RGBQUAD structure appended in recognition that the BITMAP-INFO is followed by a colour table.

```
struct /BitMapInfo
    /BitMapInfoHeader   Field bminfo.Header     \ the 40-byte header
    /RGBQUAD            Field bminfo.Colours    \ where colours go.
end-struct
```

7.5 Expanded DIB compression

Two fields in the BITMAPINFOHEADER structure which have not been discussed are bmih.Compression and bmih.SizeImage.

bmih.Compression can be one of 4 values : BI_RGB, BI_RLE8, BI_RLE4 and BI_BITFIELDS (0-3 respectively). Some values may be used only with certain pixel sizes, as follows:

1. BI_RGB : The pixel bits are stored as in the OS/2 style DIBs.
2. BI_RLE8 : The pixel bits are stored in a 'run-length encoding' (RLE) compression scheme. Used only on 8-bit DIBs.
3. BI_RLE4 : Used only on 4-bit DIBs, a 4-bit RLE compression scheme is used.
4. BI_BITFIELDS : Used in 16-bit and 32-bit EDIBs

Run-length encoding

Run-length encoding for 8-bit EBDIs handles the pixel data in pairs of bytes. The following list describes the coding scheme, with b1 being the value of the zero-th byte, b2 the value of the next etc..

1. b1=0, b2=0 : End of line.
2. b1=0, b2=1 : End of image.
3. b1=0, b2=2 : Assume the current position is (x, y), then move to the position (x+b3, y+b4).
4. b1=0, b2=3..FF : Use the next b2 pixels.
5. b1=1..FF : Repeat pixels value b2 n1 times.

Examples:

1. The following 2 bytes 05 27, translate to the five bytes : 27 27 27 27 27.
2. Case 4 above always ends on a 2-byte boundary. For example, the bytes

   00 05 45 32 77 34 59 00

   decode to

   45 32 77 34 59.
3. Cases (1) (2) and (3) above, allow some parts to be left undefined - useful in defining a non-rectangular area, and in making digital animations.
4. 4-bit DIBs RLE have similar coding: If the first byte is non-zero it is a repetition factor n, and the second byte contains two pixels which are to be repeated.
   - For example the byte sequence [07 35] (in hex) decodes to [35 35 35 3x] x here indicates that the pixel value is not yet known. If the byte pair [05 24] follows, then the whole sequence becomes [35 35 35 32 42 42].
   - If the first byte of the pair is zero, the second byte indicates the number of pixels following. Hence [00 05 23 57 10 00] decodes to [23 57 1x]. Notice the zero-padding to an even number of bytes.

7.6 Colour Masking

7.6.1 16, 24 and 32-bit DIBs with BI_RGB encoding.

The bmih.Compression field of the BitMapInfoHeader structure is also used with 16 and 32-bit EDIBs, when its value can be either BI_RGB or BI_BITFIELDS. 24-bit DIBs always have a compression value of BI_RGB, and the pixel data is a list of rows of RGBTRIPLE structures, each row padded to be a multtiple of 4.

```
Pixel   -----blue-----   -----green-----  -----red-------
        7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
```

For 16-bit EDIBs with a compression value of BI_RGB each pixel occupies 2 bytes, with the colours coded as follows:

```
Pixel   een-- ---Blue--  0 ---Red--- -Gr
        7 6 5 4 3 2 1 0  7 6 5 4 3 2 1 0
```

Each colour uses 5 bits : Blue the lowest 5 of the first byte, etc.. as above. This becomes more easily understood by accessing the 2 bytes as a 16-bit word, where the least significant byte is stored first.

```
        0 ---Red--- --Green-- ---Blue--
        1 1 1 1 1 1
        5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
```

Note that each row of a 16-bit DIB pixel data is padded to be a multiple of 4.

Colour values are extracted from the 16-bit word using shift and AND operations to give a 5-bit value. This must be multipled by 8 to give colour values in the range [0..F8].

32-bit EDIBs using BI_RGB compression stores pixel data in 32-bit 'double words' i.e. as RGBQUAD structures:

```
        |---0--||--Red-||-Green||-Blue-|
        33222222222211111111111
        10987654321098765432109876543210
```

Please notice that this 32-bit colour coding is NOT consistant with the Windows GDI function calls that use 32-bit COLORREF values. COLORREF values encode red as the least significant byte!

7.6.2 16 and 32-bit DIBs with BI_BITFIELDS compression.

In 16-bit and 32-bit DIBs with the bmih.Compression field set to BI_BITFIELDS, the BitMapInfoHeader record is immediately followed by three 32-bit colour masks - first for red then for green and then blue.

The three masks are such that no bits 'overlap' or coincide with any other - i.e. a bitwise AND between all three gives zero. The set bits in each mask must also be contiguous, and for 16-bit DIBs there is 16 bits set in total. (presumably a similar condition holds for 32-bit DIBs).

For 16-bit pixels, the colour mask bits lie in the least significant 16 bits of the 32 bit colour mask. The set bits in each mask indicate which bits in the pixel data are used for the respective colour. But in addition, the bits also specify a 'multiplication' factor by which to scale extracted colour bits. This scaling produces numbers in the range [0 .. 255].

For instance, assume red blue and green masks of $F800, $07E0 and $001F respectively. To extract red colour from (16-bit) pixel data the red mask $F800 is 'ANDed' with the pixel value and shifted right by eleven bits to gve a value of between 0 to 1F (that is any of the lowest 5 bits). This value is then to be shifted left by 3 - giving a value between 0 and 248.

Blue colour value is obtained by ANDing the pixel value with the blue mask $07E0 and shifting right by 5 to giving a value upto $3F. 3F is 6 bits, so the blue value is scaled by shifting left by 2 bits to give a value in the range [0..252] Finally, the green colour is obtained by ANDing the pixel value by 1F. There is no right shift (or there is a right shift of zero), but a left shift by 3 is then applied, thus giving values within [0..248].

The two words Mask>Rshift and Mask>Lshift, when given a mask, provide the number of bits by which to shift values and the number of bits by which to shift left. This value is the number of places to shift right which gives tha largest number less than 256. Each colour value is thus guaranteed to be within [0..255].

`: Mask>Rshift  \ u -- u'`
Return the number of bits right a colour value should be shifted after being masked with the value given. The returned value is simply a count of the least significant unset bits.

`: Mask>LShift  \ u -- u'`
Given a colour mask, return the 'left shift' value. That is the number of bits to shift left the colour value obtained after appliying the right shift given by Mask>RShift. This left shift value scales the number to be within the range [0..255].

7.6.3 The Version 4 Header

Introduced in Windows 95 (which might have been called Windows 4.0) was the BITMAPV4HEADER, replacing the BITMAPINFOHEADER. NT 4.0 also supported this structure.

```
4 constant /FxdPt2dot30     \ 32-bit fixed point size

struct /CIEXYZ
    /FxdPt2dot30    field   ciexyz.X    \ 32 bits = 4 bytes
    /FxdPt2dot30    field   ciexyz.Y    \           4 bytes
    /FxdPt2dot30    field   ciexyz.Z    \           4 bytes
end-struct                              \ Total    12 bytes

struct /CIEXYZTRIPLE
    /CIEXYZ field   ciexyz.Red      \ 12 bytes
    /CIEXYZ field   ciexyz.Green    \ 12 bytes
    /CIEXYZ field   ciexyz.Blue     \ 12 bytes
```

```
end-struct                          \ Total 36 bytes

struct /BitMapV4Header
    _dword  field   bv4.Size            \ = 120 (bytes)
    _long   field   bv4.Width           \ image width in pels
    _long   field   bv4.Height          \ height in pels
    _word   field   bv4.Planes          \ = 1
    _word   field   bv4.BitCount        \ 1,4,8,16,24 or 32
    _dword  field   bv4.Compression     \ compression code
    _dword  field   bv4.SizeImage       \ #bytes in image
    _long   field   bv4.XPelsPerMetre   \ horiz. resolution
    _long   field   bv4.YPelsPerMetre   \ vert. resolution
    _dword  field   bv4.ClrUsed         \ # colours used
    _dword  field   bv4.ClrImportant    \ # important colours
    _dword  field   bv4.RedMask         \ red colour mask
    _dword  field   bv4.GreenMask       \ green colour mask
    _dword  field   bv4.BlueMask        \ blue colour mask
    _dword  field   bv4.AlphaMask       \ Alpha mask
    _dword  field   bv4.CSType          \ colour space type
    /CIEXYZTRIPLE field bv4.EndPoints      \ XYZ values
    _dword  field   bv4.GammaRed        \ red gamma value
    _dword  field   bv4.GammaGreen      \ green gamma value
    _dword  field   bv4.GammaBlue       \ blue gamma value
end-struct
```

Notes:

- The first eleven fields are identical to those of the BITMAPINFOHEADER structure. The colour mask fields are used only for 16-bit and 32-bit DIBs when the bv4.Compression field is BI_BITFIELDS, and correspond in use and position to the colour masks descussed prevoiously.
- bv4.AlphaMask is thought not to be used.
- Remaining fields are about colour management, dealing with non-linearity of colour devices etc.
- CIEXYZTRIPLE is defined as a record of three CIEXYZ structures, each of which is a triple of 32-bit fixed point numbers in a (2.30) format (i.e. 2 bit integer part and 30 bit fraction).
- There is a little confusion here: Petzold states that this structure is 120 bytes in length (but he also states that the BITMAPV5HEADER structure is 120 bytes in length). I get a length of 108 bytes and I will stick to that for now.

7.7 Device Dependant bitmaps

Device dependant bitmaps are supported by Windows for backward compatibility reasons. New applications generally should not use them.

DDBs are described using the Windows BITMAP structure:

```
struct /BITMAP
    cell    field   bmType          \ the bitmap size. Allways zero
    cell    field   bmWidth         \ width in pixels. Must be >0
```

```
    cell    field   bmHeight        \ height in pixels. Must be >0
    cell    field   bmWidthBytes    \ number of bytes per row
    _Word   field   bmPlanes        \ number of colour planes
    _Word   field   bmBitsPixel     \ number of bits encoding a colour
    cell    field   bmBits          \ address of bit values.
end-struct
```

I have found this structure usefull for a single purpose only - finding the size of a bitmap. Calling The API function GetObject on a bitmap handle, a /BitMap structure is filled and the value of any field can be accessed.

A previous method required a device context and called the API function GetDIBits and I would guess was time consuming. The newer method (using GetObject) is also the one recomended in "Win32 Programming". (see page 358)

`: BitMap-Size       \ hBITMAP -- w h`
Return the width and height in pixels, of the bitmap whose handle is hBITMAP.

7.8 Handling Bitmap Files

7.8.1 Embedding Bitmap Files

`: BMF:  \ "name" "pathname" ++ ;`
Creates a named word in the dectionary and copies the contents of the named file to its parameter field (or whatever it is called under ANSI). Pass that address to Load-BMF to get Windows to create a bitmap and return a bitmap handle. Remember to use DeleteObject when the handle has been finished with.

`: BMF>Header    \ addr1 -- addr2`
Return the address of the bitmap info header (or core header) data, given a BMF bitmap file structure.

`: BMF>Colours   \ addr1 -- addr2`
Given a BMF bitmap file structure, return the first address of its colour table (if there is one).

`: BMF>Pixels    \ addr1 -- addr2`
Reuturn the pixel data of a Bitmap file structure.

`: CreateDisplayDC   \ -- hdc | 0`
Create a device context for the DISPLAY device. The returned value hDC is a handle to the device context which should be deleted using DeleteDC when it is finished with.

`: Load-BMF      \ bmf -- hBM`
Given the address of a bitmap file structure, get Windows to create a bitmap and return its handle.

`: Create-BM     \ hDC bmf -- hBM`
Create a bitmap which is compatible with the given device context.

7.8.2 Handling Bitmap files at run-time....

`: Load-BitmapFile   \ add len flags -- hBITMAP`
Given the name of a bitmap file, create a bitmap and return its handle. Uses the Windows API funtion LoadImage. flags are a set of LR_xxx flags.

`: Load-DIBitmap     \ add len -- hBITMAP`

Given the file name of a bitmap file, load the file data as a device independant bitmap and return a handle to it. Uses the Windows API function LoadImage.

`: Load-TransparentDIB  \ add len -- hBITMAP`
Given the file name of a bitmap file, load the bitmap data and return a handle to it. Uses the Windows API function LoadImage. The colour of the first pixel of the image is deemed to be 'transparent' : all pixels of that colour are replaced with COLOR_WINDOW. This applies only to those images which have a corresponding colour tables.

7.8.3 Miscellaneous operations on bitmaps...

`: Bitmap>DC  \ hBITMAP hDC w h --`
Draws the bitmap specified to a device context (handle *hDC*. The image will be w units wide and h units high and is placed at position (0,0) on the device context.

7.9 Icon Files.

`: Icon:  \ "name" "pathname" ++`
Create a dictionary word, and lay the contents of the given filename as its data area.

`: Create-Icon  \ 'icon n -- hICON`
Given the address of an icon resource structure create the n-th icon and return its Window handle. NOTE : the icons created have the "large icon" - which is usually 32 pixels square, even if the bits specify 16 pixels square.

7.9.1 Loading Icons from disk....

`: Load-IconFile  \ add len flags -- hICON`
Create an icon and return its Windows handle, given its file name. Calls the Window API LoadImage in which 'flags' are the LR_xxx load flags.

`: Load-Icon  \ add len -- hICON`
Create an icon and return its Windows handle, given its file name. Uses Load-IconFile.

`: Load-TransparentICON  \ add len -- hICON`
Create a transparent icon wnd return its Windows given its file name. Uses Load-IconFile

7.9.2 Miscellaneous icon handling words

`: .Icon  \ hICON -- ;`
testing word. prints icon on console screen

`: Std-IconSize  \ -- w h`
Returns the size of a standard icon.

`: small-Iconsize  \ -- w h`
Returns the size of a small icon.

7.10 Image lists

`: BitmapList:  \ "name" "filepath" ++`
Create a structure which can be used to create an image list. The bitmap is assumed to consist of one or more _square_ images of sides equal to the bitmaps height. Thus a bitmap which is 16 pixels high and 48 pixels wide is considered to be 3 images, each 16 pixels hight and 16 pixels wide.

`: Bitmap>ImageList  \ hBITMAP -- hImageList`
Given a bitmap handle, use it to create an image list. If w and h are the width and height of the bitmap, the image list is created to consist of (w/h) square images each of side h pixels.

```
: Load-BMList   \ 'BitmapList -- hIMAGELIST
```

Given a BMF bitmap structure, create a new image list based on that bitmap.

8 Combo boxes

8.1 Simple Combo boxes

Combo Box controls are one of the six Windows predefined controls. (The others being Button, Edit, Listbox, Scrollbar and static controls).

Windows supplies three types of combo box : *Simple*, *drop-down* and *drop-down list boxes*. These types are defined below.

`WinControl: ComboBox "combobox"`
A simple combo box having the CBS_SIMPLE style.

`Another ComboBox DropDownCombo`
A drop down list in which the selection field is an edit box and may contain text which is not contained in the list.

`Another ComboBox DropDownListCombo`
A drop down combo box in which the edit field contents must be a member of the list.

8.2 Simple Combo box commands.

The following is a glossary of the Combo Box manipulation words. The following notation is used for stack parameters:

hCombo The handle of a combo box control

add len Address and length of a text string

caddrz Address of a zero-terminated string

Other synbols have the usual meaning and may be described further in the word's description.

Many word names contain either *combo* or *CBitem* as a prefix or a suffix. *combo* is used to indicate an operation on the combo box 'as a whole' while *CBitem* is used to refer to a particular item in the combo box's drop-down list.

```
: cbCount       \ hCombo -- n
```
Returns the number of items in a combobox.

```
: cbDelete          \ index hCombo --
```
Delete the given item from the combo box.

```
: cbClear       \ hCombo --
```
Deletes all items/strings in a combobox.

```
: cbAddStrz     \ caddrz hCombo --
```
Add the zero terminated string caddrz to the combo box.

```
: cbAddStr      \ add len hCombo --
```
Add the string given by the add len pair to the combo box.

```
: cbInsertz     \ caddrz hCombo n --
```
Insert the z-string into the combo list at position n.

```
: cbInsert      \ caddr ulen hCombo n1 -- n2 ;
```

Insert the text *caddr ulen* into the list of the combo box with handle *hCombo*. The text is inserted at the *n1-th* position in the list. *n2* is the position at which the text was inserted.

`: cbSetValue \ n1 n2 hCombo --`
Store a 32-bit value n1 at position n2 in the Combo box whose handle is *hCombo* .

`: cbValue \ n2 hCombo -- n1`
Retrieve the 32-bit value n1 from position n2 in the combobox

`: cbAddPair \ add len val hCombo --`
Add the two items given by add/len (a string) and val (any 32 bit value), to the combo box whose handle is hCombo

`: cbAddStrings \ addr n hCombo --`
Copy the *n* counted strings starting at *addr* to the combo box whose handle is *hCombo*. The strings will probably have been compiled into the dictionary using `",`. In any case, each string is assumed to lie on an `aligned` boundary, and succesive strings are reached using the phrase *\co{count + aligned}. pairs with the first cell holding the number of items. The items in this table are added to the combo box whose handle is hCombo. The table is typically compiled into the dictionary as follows:

```
Create ComboPairs
```

`: cbTextLen \ idx hCombo -- len`
Return the length of the text for the idx-th item in the list

`: (Get-CBText) \ n hCombo addr -- addr len'`
Get the text of the n-th item of combo box drop down list whose handle is hCombo. Enough space should have been allocated (using CB-GETLBTEXTLEN perhaps) to receive the string. The number of chars is returned. If an error occured len' is equal to CB_ERR.

`: cbCopyText \ n hCombo add len -- add len'`
Copy the text of the n-th item of combo box whose handle is given to the buffer at address "add" and size "len". No more than "len" characters/bytes will be read in. The actual number of characters copied is returned as len'

`: cbSelect \ n hCombo --`
Select the n-th combo item

`: cbSelected \ hCombo -- n|-1`
Return the currently selected combobox item or -1 if no item is selected.

`: cbSelectedVal \ hCombo -- value`
Return the value assigned to the currently selected combobox item. A value of -1 is returned if no item is selected.

`: cbSelectVal \ val hCombo --`
Select the item whose value is val.

`: cbSelectNearest \ val hCombo --`
Select the item whose value is val or greater.

`: cbFindz \ caddrz hCombo -- n|-1`
Search the combo box list for a case insensitive match of the given zero terminated string.

`: cbFindStr \ caddr len hCombo -- n|-1`
Find the given text in the combo box list and return its index number or -1 if the text is not found. Uses cbFindz.

`: cbSelectStr \ caddr len hCombo -- n|-1`

Find the given text in the combo box list and select it. Return its index number or -1 if the text is not found.

```
: cbSetText     \ caddr len hCombo --
```
Select the given text from the list of items in the combo box. If the text doesn't exist, the window text is set instead.

```
: cbTextz   \ hCombo -- caddr
```
Returns the text of a combo box. This routine uses the Windows WM_GETTEXT routine but ignores the returned value which can be incorrect - at least for DropDownCombo boxes. The address returned is limited to 64 characters, including the termination null.

```
: cbText    \ hCombo -- caddr u
```
Returns the text of the combo box of at most 63 characters.

```
: cbExpand      \ hCombo --
```
Show the drop-down lists box of a combo box with either the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

```
: cbContract    \ hCombo --
```
Hides the drop-down lists box of a combo box with either the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

```
: cbInitStrings     \ table hCombo --
```
Initialises a combo box whose handle is hCombo. Table is the address of a list of counted strings : the first cell is the number of strings in the table. The strings follow and are cell alligned.

```
: cbInitPairs    \ table hCombo --
```
Initialises a combo box whose handle is hCombo. Table is the address of a list of counted string/32-bit value pairs : the first cell is the number of string/value pairs in the table. The item pairs follow and are cell alligned.

9 Listbox Controls

9.1 Simple Listboxes

ListBox controls are one of the six Windows predefined controls. (The others being Button, Edit, Combo Box, Scrollbar and static controls).

`WinControl: ListBox "listbox"`
The standard listbox control.

9.2 Standard listbox operations

`: lbCount \ hLB -- n`
Returns the number of items in a list box.

`: lbDeleteItem \ n hLB --`
Delete item number n in the given list box.

`: lbClear \ hLB --`
Deletes all items/strings in a list box.

`: lbAddStrz \ $z hLB --`
Append the zero terminated string to the given list box.

`: lbAddStr \ add len hLB --`
Appends the text given in by add/len to the list box.

`: lbValue! \ val hLB idx --`
Set the value of the *i\{idx}-th list box item to the value *val*

`: lbValue@ \ hLB n1 -- n2`
Retreive the 32-bit value n2 from position n1-th item of the listbox whose handle is hLB.

`: lbAddPair \ add len val hLB --`
Add the string given by the add len pair to the combo box instance given.

`: lbSelected \ hWnd -- idx|-1`
Return the currently selected list box item. The returned value is undefined for list boxes with the LBS_MULTIPLESEL and LBS_EXTENDEDSEL Styles.

`: lbSelect \ idx hWnd --`
Set the currently selected item it *idx*. For multi-select list boxes use `lbSelect+`.

`: lbSelect+ \ hWnd idx --`
Select item with index *idx* on a multi-select list box.

`: lbSelect- \ hWnd idx --`
Deselect item with index *idx* on a multi-select list box.

`: lbSelected? \ hWnd idx -- flag`
Returns TRUE if the *idx*-th item of a list box (with handle *hWnd*) is selected.

`: lbSelCount \ hWnd -- n`
Returns the number of items selected from the list box whose handle is *hWnd* which must be a multi-select list box.

`: lbTextLen \ idx hLB -- len`
Returns the length of the text at the idx-th position

```
: lbCopyText   \ n hlb add len -- add len'
```
Get the text of the n-th item of list box whose handle is hLB. No more than len characters/bytes will be read in. The actual number of characters read is returned as len'

```
: lbItemText    \ n hLB -- caddr len
```
Return the text of the n-th listbox item. caddr is the address of a static buffer in the dictionary of size MAX_PATH bytes.

```
: lbLineHt      \ hLB -- n
```
Return the line or item height or a list box. Used only if the list box does not have the LBS_OWNERDRAWVARIABLE style.

```
: lbLineHt!     \ n hLB --
```
Set the line height (the height of each item) for the list box. Used only if the list box does not have the LBS_OWNERDRAWVARIABLE style.

```
                { | r[ rect ] -- }
```
Returns the column width in a multicolumn list box.

```
: lbColWidth!   \ n hLB --
```
Sets the column width in a multicolumn list box. (A multicolumn list box has the LBS_MULTICOLUMN style).

```
: lbItemAt  \ x y hLB -- n ;
```
Return the index of the item nearest to the position x,y

```
: lbFindPrefix      \ ca u n hLB -- index|-1
```
Return the index of the next occurance of a string prefix starting from the n-th item in the list.

```
: lbFindString      \ ca u n hLB -- index|-1
```
Return the index of the next occurance of a string starting from the n-th item in the list. The search is for an exact match.

```
: lbNextPrefix      \ ca u hLB -- index|-1
```
Search for the next item which starts with the given text, starting from the item after the currenty highlighted/selected item.

10 Rich Edit Controls

Rich edit controls are enhanced multi-line edit controls allowing larger amounts of data to be handled, formatting capabilities and other enhacenments.

Most of the *Edt-xxx* words will operate on rich edit controls. The operations in this wordset are prefixed by *REdt-* and must not be used on multi-line edit controls.

10.1 Rich Edit Notification messages

Rich edit controls pass notifcation messages (i.e. WM_NOTIFY) to their parents in the same way that edit controls do and most of the edit control notification messages are supported.

A notification message passes two items of information :

- wParam : the ID of the control which originated the message.
- lParam : a pointer to a NMHDR structure or an extended NMHDR structure.

A NMHDR structure contains 3 fields :

- hwndFrom \ handle of control sending the message
- ifFrom \ ID of control sending the message
- code \ the notification code.

In VFX these fields are termed NMHDR.HWNDFROM NMHDR.IDFROM and NMHDR.CODE

The following notification messages are available for rich edit controls.

- **EN_CHANGE** - The user has modified text in an edit control. Windows updates the display before sending this message (unlike EN_UPDATE).
- **EN_CORRECTTEXT** - a SYV_CORRECT gesture occurred (whatever that is).
- **EN_DROPFILES** - the user is attempting to drop files into the control. Sent when a WM_DROPFILES message is received.
- **EN_ERRSPACE** - The edit control cannot allocate enough memory to meet a specific request.
- **EN_HSCROLL** - The user has clicked the edit control's horizontal scroll bar. Windows sends this message before updating the screen.
- **EN_IMECHANGE** - the IME conversion status has changed.
- **EN_KILLFOCUS** - The user has selected another control.
- **EN_MAXTEXT** - While inserting text, the user has exceeded the specified number of characters for the edit control. Insertion has been truncated. This message is also sent either when an edit control does not have the ES_AUTOHSCROLL style and the number of characters to be inserted exceeds the width of the edit control or when an edit control does not have the ES_AUTOVSCROLL style and the total number of lines to be inserted exceeds the height of the edit control.
- **EN_MSGFILTER** - A keyboard or mouse event has occurred.
- **EN_OLEOPFAILED** - A user action on an OLE object has failed.

- **EN_PROTECTED** - the user is taking an action that would change a protected range of text.
- **EN_SAVECLIPBOARD** - the control is closing and the clipboard contains information.
- **EN_SETFOCUS** - The user has selected this edit control.
- **EN_STOPNOUNDO** - An action occurred for which the control cannot allocate enough memory to maintain the undo state.
- **EN_UPDATE** - The user has altered the text in the edit control and Windows is about to display the new text. Windows sends this message after formatting the text, but before displaying it, so that the application can resize the edit control window.
- **EN_VSCROLL** - The user has clicked the edit control's vertical scroll bar. Windows sends this

Controlling Notification Messages

An application has some control over which notification messages a rich edit control will send. Specified bits in a rich edit controls *Event Mask* determine which message may sent. The messages controled by a rich edit event mask are:

- ENM_CHANGE Sends EN_CHANGE notifications.
- ENM_CORRECTTEXT Sends EN_CORRECTTEXT notifications.
- ENM_DROPFILES Sends EN_DROPFILES notifications.
- ENM_KEYEVENTS Sends EN_MSGFILTER notifications for keyboard events.
- ENM_MOUSEEVENTS Sends EN_MSGFILTER notifications for mouse events.
- ENM_PROTECTED Sends EN_PROTECTED notifications.
- ENM_REQUESTRESIZE Sends EN_REQUESTRESIZE notifications.
- ENM_SCROLL Sends EN_HSCROLL notifications.
- ENM_SELCHANGE Sends EN_SELCHANGE notifications.
- ENM_UPDATE Sends EN_UPDATE notifications.

By default the event mask is zero. Windows provides the EM_GETEVENTMASK and EM_SETEVENTMASK messages to manipulate the event mask. The following words are provided simplyfy this task.

`: REdt-EMask@  \ hWnd -- u`
Returns the current value of a rich edit control's event mask.

`: REdt-EMask!  \ u hWnd --`
Set the event mask of a rich edit control whose handle is *hWnd*.

`: REdt-EMask+  \ u hWnd --`
Add the event bit mask u to the rich edit control whose handle is *hWnd*.

`: REdt-EMask-  \ u hWnd --`
Add the event bit mask u to the rich edit control whose handle is *hWnd*.

The MSGFILER data structure.

Mouse and keyboard events produce EN_MSGFILTER notification messages (in the case that the event mask bits ENM_KEYEVENTS and ENM_MOUSEEVENTS are set).

The lParam value for EN_MSGFILTER notification message is the address of an extended NMHDR data structure, defined as follows:

```
struct /MSGFILTER
    NMHDR +
    cell    Field   MSGF.msg
    cell    Field   MSGF.wParam
    cell    Field   MSGF.lParam
end-struct
```

The fields have the following meanings:

- **NMHDR** - a NMHDR structure. The code member of the NMHDR structure is the EN_MSGFILTER notification code that identifies the message being sent.
- **msg** - the keyboard or mouse message identifier.
- **wParam** - the wParam parameter of the message.
- **lParam** - the lParam parameter of the message.

Other notification messages use different extended NMHDR data structures - see your Windows documentation for details.

10.2 Manipulating Text

Most of the edit control messages are supported by rich edit controls, and so the various *edt-xxx* words which apply to edit controls also apply to rich edit controls. (See the chapter about *Controls*).

```
: REdt-ToEnd   \ hCtl --
```

Put caret at end of the text.

```
: REdt-Select   \ start len hWnd --
```

Select the indicated text in a rich edit control. *start* is the zero-base index into the control's text and *len* is the number of characters to select.

```
: REdt-Appendz  \ addrz hCtl --
```

Append the zero-terminated text to the existing rich edit control's text.

```
: REdt-Append   \ add len hCtl --
```

Append text (add len) to the edit control whose handle is hCtl.

```
: REdt-Selected    \ hRichEd -- start len
```

Return the starting position and length of the selected text in the rich edit control

10.3 The RichEdit control

```
WinControl: RichEdit "RichEdit"
```

The Rich Edit control. It has the typical WS_xxx styles of a control and WS_HSCROLL and WS_VSCROLL in addition. The edit control specific styles are ES_MULTILINE and ES_WANTRETURN and ES_AUTOVSCROLL to give the expected window-editing behaviour. See below for more about control styles.

```
Another RichEdit RichEdit-r/o
```

A *Read Only* rich edit control, similar to `RichEdit` above but with the ES_READONLY Style.

10.4 Rich edit control styles

The `RichEdit` control defined above has been assigned styles to allow the typical behaviour one might expect from an edit control. `RichEdit-r/o` provides a similar display but the text may not be changed by the user.

`RichEdit` has the ES_AUTOVSCROLL style. This causes text to be scrolled vertically whenever necessary. But there is a similar style ES_AUTOHSCROLL, which causes text to be scrolled *horizontally*. `RichEdit` does not have this style and lines of text are 'wrapped' around onto the next line if their length is wider than the control's client area.

A rich edit control with the "word wrap" property will not always be suitable. A program editor for instance, better not put what appear to be "line-end" characters here and there. To remove the word wrap behaviour add the ES_AUTOHSCROLL style.

The following example is a dialoog box containing two rich edit controls identical in every way except that one of them has the ES_AUTOHSCROLL style. Open the dialog box to illusrate the differences between the two types of control.

10.5 An Example

```
another RichEdit RichEdit/WW+   \ A rich edit with wordwrap
    ES_NOHIDESEL Style+
    es_autohscroll  Style-
end-control


another RichEdit RichEdit/WW-   \ A rich edit without wordwrap
    ES_NOHIDESEL Style+
    es_autohscroll  Style+
end-control

NextID: IDC_REdit/WW+
NextID: IDC_REdit/WW-

: appz",        \ "text" --
    1 chars negate allot
    z\",
;


create REtext
    z\", This first line is too long to fit on a single line.\n"
    appz", This is line two\nand line three"


Dialog: REBox
    caption "Rich Edit box"
    150 100 Size
    WS_VISIBLE WS_CAPTION or WS_SYSMENU or Style
    DS_CENTER Style+
    WS_EX_CONTROLPARENT ExStyle+
```

```
    10 600 0 Font "MS Sans Serif"

    0 0 75 10   IDC_STATIC      z" Word wrap on"    TextLabel
    0 10 75 90  IDC_REdit/WW+   REtext              RichEdit/WW+
    75 0 75 10  IDC_STATIC      z" Word wrap off"   TextLabel
    75 10 75 90 IDC_REdit/WW-   REtext              RichEdit/WW-

end-dialog

: Show-REBox
    0 REBox ShowModeless Set-CurrWin
;
Show-REBox
???
```

11 Scrollbar helpers

11.1 Introduction

When viewing documents which are too large to fit within a window, scroll bars are often provided to allow users to scroll the data bringing other parts into view.

These *standard scroll bars* are encountered lying either vertically along one side of a window or horizontally along the bottom of a window. Standard scroll bars are part of a window and are positioned outside its client area.

A second type of scroll bar is the *scroll bar control* which may be positioned anywhere within another window just like any control.

This wordset provides a means of manipulating each type of scroll bar with names chosen to indicate the scroll bar type. The following word prefixes are used to indicate the type of scroll bar:

- **VScroll** - vertical standard scroll bar
- **HScroll** - horizontal standard scroll bar
- **SBar** - scroll bar control

11.2 Glossary

Scroll bar position

`scbar-Pos  \ hWnd -- pos`
Returns the position of a scroll bar control.

`HScroll-Pos  \ hWnd -- pos`
Returns the position of a horizontal scroll bar.

`VScroll-Pos  \ hWnd -- pos`
Returns the position of a vertical scroll bar.

`scbar-SetPos  \ pos hWnd --`
Sets the position of a scroll bar control.

`HScroll-SetPos  \ pos hWnd --`
Sets the position of a horizontal scroll bar.

`VScroll-SetPos  \ pos hWnd --`
Sets the position of a vertical scroll bar.

The scroll bar range.

A scroll bar's range is its minimum and and its maximum positions.

`scbar-Range  \ hWnd -- min max`
Returns the range of a scroll bar control.

`HScroll-Range  \ hWnd -- min max`
Returns the range of a horizontal scroll bar.

`VScroll-Range  \ hWnd -- min max`

Returns the range of a vertical scroll bar.

`scbar-SetRange  \ min max hWnd --`
Sets the range of a scroll bar control.

`HScroll-SetRange  \ min max hWnd --`
Sets the range of a horizontal scroll bar.

`VScroll-SetRange  \ min max hWnd --`
Sets the range of a vertical scroll bar.

Enabling and Disabling a scroll bar

`scbar-Disable  \ hWnd --`
Disable a scroll bar control.

`HScroll-Disable  \ hWnd --`
Disable a horizontal scroll bar.

`VScroll-Disable  \ hWnd --`
Disable a vertical scroll bar.

`: scbar-Enable  \ hWnd --`
Enable a scroll bar control.

`: HScroll-Enable  \ hWnd --`
Enable a horizontal scroll bar.

`: VScroll-Enable  \ hWnd --`
Enable a vertical scroll bar.

Showing & Hiding scroll bars

`: scbar-Hide  \ hWnd --`
Hide a scroll bar control.

`: HScroll-Hide  \ hWnd --`
Hide a horizontal scroll bar.

`: VScroll-Hide  \ hWnd --`
Hide a vertical scroll bar.

`: scbar-Show  \ hWnd --`
Show a scroll bar control.

`: HScroll-Show  \ hWnd --`
Show a horizontal scroll bar.

`: VScroll-Show  \ hWnd --`
Show a vertical scroll bar.

Setting and Getting percentage values...

`: scbar-Percent  \ hWnd -- u`
Return the position of a scroll bar control as a percentage of its range.

`: HScroll-Percent  \ hWnd -- u`
Return the position of a horizontal scroll bar as a percentage of its range.

`: VScroll-Percent  \ hWnd -- u`
Return the position of a vertical scroll bar as a percentage of its range.

```
: scbar-SetPercent   \ pc hWnd --
```
Set the position of a scroll bar control expressed as a percentage of its range.

```
: HScroll-SetPercent    \ pc hWnd --
```
Set the position of a horizontal scroll bar expressed as a percentage of its range.

```
: VScroll-SetPercent    \ pc hWnd --
```
Set the position of a vertical scroll bar expressed as a percentage of its range.

Setting and Getting the size of the scroll box (thumb)

For some reason the movable button on a scroll bar known as a "scroll box" is sometimes known as the "thumb". Its position within the scroll bar represents the position value of the scroll bar.

In Windows V4.0 and later the size of the scoll box can be changed. It is usually set so that its size as a fraction of the lenngth of the scroll bar equals the amount of document shown as a fraction of the length of the whole document. The scroll box size is termed *page size*.

The following words help in manipulating a scroll bar page size. The words have the posfix "boxsize" rather than "pagesize".

```
: scbar-SetBoxSize       \ n hCtl --
```
Set the scroll bar control's box size to n.

```
: HScroll-SetBoxSize    \ n hCtl --
```
Set the horizontal scroll bar's box size to n.

```
: VScroll-SetBoxSize    \ n hCtl --
```
Set the vertical scroll bar's box size to n.

```
: scbar-BoxSize          \ hCtl -- n
```
Retrieve the scroll bar control's box size.

```
: HScroll-BoxSize       \ hCtl -- n
```
Retrieve the horizontal scroll bar's box size.

```
: VScroll-BoxSize       \ hCtl -- n
```
Retrieve the vertical scroll bar's box size.

12 Statusbar Controls

12.1 Introduction

Statusbars are those familiar areas at the bottom of windows, divided into sections each of which shows some application data. A Statusbar can be positioned at the top of a window rather than the bottom and can show a 'sizing grip' at the right hand end.

A Statusbar is termed 'Status Window' in Windows.

12.2 The Statusbar Controls

A Statusbar is one of the Windows Common Controls. This means that the Windows function InitCommonControls must be executed before a statusbar can be used to ensure the correct OS library is loaded. (VFX does this by default).

Information about a statusbar is held in its WinProps. In particular an array of information is maintained for each section and for this reason a maximum number of sections is defined.

`20 constant MaxStatusParts`

The maximum number of extions which may be created for each statusbar. An array of this size is created in a statusbar's window properties, and this number was chosen to limit the size of that array. There is no other reason for this limit which may be changed if need be.

`WinProps: /StatusBar`

The statusbar window properties structure.

`WinControl: StatusBar "msctls_statusbar32"`

The standard statusbar control. It will be positioned at the bottom of its parent window and has no sizing grip area.

`Another StatusBar GripStatusBar`

Another statusbar which displays a sizing grip at the right hand side.
The following Common Control Styles may be used to define more statusbars:

- **CCS_BOTTOM** - Default style - places control along bottom of its parent window.
- **CCS_NOPARENTALIGN** - stops auto sizing??
- **CCS_NORESIZE** - stops auto sizing??
- **CCS_TOP** - positions at top of control rather than the bottom.
- **SBARS_SIZEGRIP** - adds the 'size grip' area to the control. (don't use with CCS_TOP style)

12.3 Statusbar parts

Statusbars can be divided into sections which Windows calls 'parts'. The following words provide for creating and sizing those parts followed by setting those part's text.

Creating status window parts.

The following program fragment shows how to create a statusbars parts:

```
IDC_STATUSBAR CtlHdl    \ get handle to the control
dup stbClear            \ starts with no parts
50 over stbAddPart      \ create a part with a width of 50
74 over stbAddPart      \ another one with a width of 70
etc....
stbAssemble             \ create the parts
```

`: stbClear    \ hWnd --`
Sets an internal parts counter to zero.

`: stbAddPart     \ n hWnd --`
Sets up a new part which will have a width of n pixels. This sets the statusbar internal data only - the statusbar will not contain this section until `SBar-SetParts` is executed.

`: stbAssemble   \ hWnd --`
Creates the parts in Windows.

`: stbCount      \ hWnd -- n ; get number of parts`
Returns the number of parts stored in the internal counter (i.e. stored in the statusbar's WinProps)

`: stbEqualise   \ hWnd --`
Set all parts to be equally spaced along its length.

`: stbResetPart    \ width hWnd idx --`
Resizes a parts width. This is done by adjusting the internal data and then executing `stbAssemble`.

`: stbSetText      \ caddr ulen hWnd n -- ;`
Set text for part nunber n. A border is drawn around the text making it appear below the plane of the window.

`: stbSetPopout      \ caddr ulen hWnd n -- ;`
Set popout text for part n. The text is drawn to appear above the plane of the window.

`: stbOwnerDraw    \ 32bit hWnd n --`
Sets part n to be an 'Owner drawn' part - Windows sends WM_DRAWITEM messages to the parent window for this part. Note that in GUIgen both windows and dialog boxes reflect WM_DRAWITEM messages to the child control as GWM_DRAWSELF messsages giving a control the ability to 'draw itself'.

`: stbText       \ hWnd n -- caddr ulen`
Returns the text of part number n.

`: stbInPart     \ x y hWnd n -- flag`
Return TRUE if the coordinates x and y fall within the bounding rectangle if the n-th part.

13 GrabBar Controls

13.1 Introduction

A Grab Bar is a long thin Window control allowing users to 'grab' it by clicking the left mouse button while over the control. While holding the button down users can 'move' the control to another point within the parent window.

Grab bars provide a way of users resizing controls by dragging their 'edges' rather like the division between the two sides in Windows Explorer can be moved.

Two types of grab bar are defined. Vertical grab bars lie down a window and can be moved left or right. Horizontal grab bars lie accross a window and can be moved up or down.

The 'private' code is compiled into a VFX module called GrabBars. This code may be accessed using the phrase

```
expose-module grabbars
```

13.2 Grab Bar Messages

The following messages are defined to comunicate with a grab bar's parent window.

`AppMessage: GBM-AskLimits`
This message is sent to a grab bar's parent window during its mouse down event to request the area within the parents client area that the mouse should be restricted to. wParam is the ID of the control and lParam is a RECT structure containing the default limits which should be changed if required. There is no return value.

`AppMessage: GBM-AskMove`
A message sent to a grab bar's parent window when the mouse has been released asking for 'permission' to move a grab bar to its new location. A returned value of TRUE causes the grab bar to be moved. wParam is the ID of the grab bar control. lParam is the address of a rectangle containing the proposed grab bar outline coordinates. The message handler is free to alter the rectangle coordinates to force the bar to be draw in any position required.

If the grab bar has the GBS-AutoMove style this message is not sent.

`AppMessage: GBM-TellMove`
A message sent to the control's parent window when it is moved to a new position (after having grabbed the bar and then releasing it). wParam is the ID of the grabbar. lParam is the address of a rectangle containing the new grab bar outline coordinates.

`AppMessage: GBM-SWAPINFO`
A message to enable information to be sent to or taken from a Grab Bar. Introduced January 2010 to allow the background colour of a grab bar to be set by an application. lParamLo is the function number which specifies what data is being transferred. wParam and lParamHi are function number specific.

`1 constant GBS-AutoMove`
A grab bar style which causes the grab bar to automatically after having grabbed the bar. When this style is set the GBM-AskMove is not sent.

`: GB-SETBKGD    \ colour hWin --`
Set the background colour of the grab bar whose handle is hWin to the specified colour. If colour is negative, a default colour of COLOR_BTNFACE is used.

`GrabBarClass AppControl: VGrabBar`
A vertical grab bar with the GBS-AutoMove style

`GrabBarClass AppControl: HGrabBar`
A horizontal grab bar with the GBS-AutoMove style

13.3 A Grab Bar Example

The following code shows two grab bars being used within a single dialog box. One has the `GBS-AutoMove` style (the default condition) and one doesn't (it has been taken away).

```
NextID: IDC_Vert        \ ID of the vertical bar
NextID: IDC_Horz        \ ID of the horizontal bar

: gbd-AskLimits     \ --
\ GBM-AskLimits handler for vertical and horizontal bar. The
\ client area coordinates, which are supplied as the default
\ coordintes as a RECT at 'lParam', are modified to 'move away
\ from' the edges of the dialog box.
    10 lParam rect.left +!
    10 lParam rect.top +!
    10 lParam rect.right -!
    10 lParam rect.bottom -!
;

: gbd-AskMove   \ --
\ GBM-AskMove handler. Simply returns a TRUE value to 'give
\ permission' for the move and doesn't alter the new position
\ coordinates held in the rectable at lParam.
    TRUE ReturnVal !
    -inherited  \ 001:16-Mar-06 Do-DefWndProc? off
;

Another VGrabBar VertGrabber
    GBS-AutoMove Style-
end-control

dialog: GrabBarDlg
\ The test dialog box.
    caption "Statusbar test"

    200 75 Size

    WS_VISIBLE WS_CAPTION or Style
    WS_THICKFRAME Style+
    WS_SYSMENU Style+
    DS_CENTER Style+
    DS_SetFont Style+
```

```
    10 100 0 Font "MS Sans Serif"

    10 10 2 50  IDC_Vert    z" vert"    VertGrabber
    40 50 40 4  IDC_Horz    z" horz"    HGrabBar

    GBM-AskLimits   does gbd-AskLimits
    GBM-AskMove     does gbd-AskMove

end-dialog


: Go    \ --
\ Shows the test dialog box.
    0 GrabBarDlg showmodeless Set-CurrWin
;
go

[then]
```

14 Column Lists

14.1 Introduction

A Column List is a control used to display a list if items in a grid-like way. A single row corresponds to one item as does the standard Window listbox control. However, unlike a listbox, the Column List control can be used to show data in more than one column.

Also unlike a list box, the data are NOT held within the control - when any datum is required the control sends a message to its parent window asking for it. These data are obtained when they are to be 'painted' (i.e. displayed within the column list control), and are left-aligned, right-aligned or centred within a column. A list of column alignment values is held internally for this and these may be changed by an application at run-time.

The Column List control is based on an original window class (not a sub-class of an existing control), and is 'topped' by a Windows Header control. There should be no need to program the header control directly - for instance, setting the width of a column also sets the header section width.

Scroll bars are also an integral part of this control. They are visible only when the data do not fit into the control area.

As of November 2007 a column list which has been given the `CLS-MultiSelect` style is set to *multi-select* mode, in which more than one line can be hightlighted using the mouse with the Control and ALT keys.

Word names

In naming data structure fields, it appears to be MPE's practice to prefix field names with the name of the structure followed by a full stop. I often follow this practice and extend it by prefixing data operators with the structure name followed by a dash.

Ever the lazy programmer, I have shortened prefixes here to the three letters : **cl.** prefixes a data structure element, and **cl-** prefixes an operator. None-public words which are 'hiddden' in the COLUMNLISTS vocabulary may not be burdened with a prefix.

Requirements

Before the ColumnList control can be compiled, the standard GUIgen files, SCROLLS.FTH and HdrCtrl.fth should be loaded.

14.2 Glossary

Setting up

Since I have never built anything which didn't require later tamporing, a vocabulary is used to contain and hide the internals rather than a MODULE. A MODULE 'seals' code internals rather better and one might be used in the future but for now the ease of using the `ONLY/ALSO` words is more important.

The 'high level' words, those for use by applications are `EXPORTED` to the Forth vocabulary and a list of those words appears towards the end of this document.

`vocabulary ColumnLists`
The Column List words are compiled into this vocabulary.

Application defined window messages

The following Window messages are defined for use with the column list control. Messages which are 'sent to itself' can be detected by sub-classing a column list control using ANOTHER.

`AppMessage: CLM-ROWCHANGED`
The message sent when a selected row is changed either by a mouse click or a key press. If the CLS-INFORMSELF window style has been specified the message is sent to itself. Otherwise the message is sent to the control's parent window.

wParam is the handle of the control. lParam = new row number.

`AppMessage: CLM-DBLCLICKED`
This message is sent to the controls parent when a row is 'selected' by double-clicking with the mouse. If the CLS-INFORMSELF window style has been specified the message is sent to itself.

wParamLo is the ID of the handle of the column list control. wParamHi is undefined and lParam is the address of a /CLDBLCLICK structure.

`AppMessage: CLM-SORTCOLUMN`
Sent when a mouse click occurs on a header control button. The message is sent to the parent window unless the control has the CLS-INFORMSELF style in which case it is sent to itself. This message is typically used to let the user signify that the items in the list should be ordered according to the column which was clicked.

wParam is the handle of the control. lParam = column number.
If a non-zero result is returned, the column list will be repainted.

`AppMessage: CLM-ASKSTRING`
A message sent by a Column List to request the string value at a particular row and column. Note that this message is sent either to the parent or to the control itself depending on the control style CLS-ASKSELF. wParam is the columnlist window handle. lParam is an AskString structure.

`Appmessage: CLM-DRAWCELL`
A message sent to column lists with the CLS-DRAWSELF style. wParam is unused (zero) and lParam is the address of a /DRAWCOLLISTSTRUCT structure (see below).

`AppMessage: CLM-SETPROP`
A message used to set various column list properties such as setting the number of columns, rows etc. wParam is the value of the property being set. lParamLo defines the property. A complete set of words to set these properties is provided so application should not need to handle this message. The following wParam and lParam values are specified:

- lParamLo = 0 (Set number of columns), wParam is the number of columns.
- lParamLo = 1 (Set number of rows), wParam is the number of rows.
- lParamLo = 2 (Set column width), wParamLo is column number, wParamHi is width.
- lParamLo = 3 (Clear/Empty the list), wParam is unspecified.

- lParamLo = 4 (Set equal column widths), wParam is unspecified.
- lParamLo = 5 (Set current row), wParamLo is row number.
- lParamLo = 6 : Select Row. wParamLo is row number
- lParamLo = 7 : Deselect Row. wParamLo is row number
- lParamLo = 8 : wParam=0:Select all, wParam<>0 Deselect all.
- lParamLo = 9 : Spread columns to fit control's width
- lParamLo = 10 : Set Line/row Height
- lParamLo = 11 : not implemented
- lParamLo = 12 : not implemented
- lParamLo = 13 : not implemented
- lParamLo = 14 : Set/Reset column mode
- lParamLo = 15 : Set the current column
- lParamLo = 16 : Set the column text alignment
- lParamLo = 17 : not implemented
- lParamLo = 18 : Set the top row number
- lParamLo = 19 : Set minimum column width
- lParamLo = 20 : Set minimum Header Height

`AppMessage: CLM-GETPROP`
A message used to receive various columnlist property values. lParamLo is a number representing the property in question. lParamHi and wParam are specified depending on lParamLo and unless otherwise stated below, these values are not used. The value returned by SendMessage is the value required.

- lParamLo = 0 : Get number of columns.
- lParamLo = 1 : Get number of rows.
- lParamLo = 2 : Get column width. wParamLo is column number.
- lParamLo = 3 : unspecified.
- lParamLo = 4 : Get sum of column widths
- lParamLo = 5 : Get current row.
- lParamLo = 6 : Test Row Selected. wParamLo is row number
- lParamLo = 7 : not implemented
- lParamLo = 8 : get number of selected rows
- lParamLo = 9 : not used
- lParamLo = 10 : Get Line/row Height
- lParamLo = 11 : convert y-value to row#. lParamHi=y value. wParamLo = 0
- lParamLo = 11 : convert x-value to col#. lParamHi=y value. wParamLo = 1
- lParamLo = 12 : Return the header control handle
- lParamLo = 13 : Used to get string for column wParamlo, row wParamHi
- lParamLo = 14 : Returns TRUE if in column mode
- lParamLo = 15 : Returns the current column
- lParamLo = 16 : Get the column text alignment
- lParamLo = 17 : Get the given cell's rectange coordinates
- lParamLo = 18 : get top row number
- lParamLo = 19 : Get minimum column width

- lParamLo = 20 : Get minimum Header Height

`AppMessage: CLM-CHKControl`
A message used to check or query various properties of a column list to provide a way of forcing a column list to check various properties according to the value of wParam. Applications will use the 'wrapper' words rather than this message.

wParamLo is a code number specifying the action to take.

- wParamLo = 0 : Check visibility of horizontal scroll.
- wParamLo = 1 : Check visibility of vertical scroll.
- wParamLo = 2 : Check visibility of both scroll bars.
- wParamLo = 3 : Check columns widths and header position
- wParamLo = 4 : Check header size

Column list styles

`ColListStyle: CLS-ASKSELF`
A subclassed column list with this style will receive CLM-ASKSTRING messages, so allowing it to control its own display. If CLS-ASKSELF is not set as a style value the message is sent to the control's parent.

wParam is the handle of the control, lParam is the address of a /ASKSTRING data structure described below. The structure contains the row and column numbers being considered in the askstring.row and askstring.col fields. The application supplies the address and length of the requested string in the askstring.text and askstring.len fields. The /ASKSTRING structure

```
struct /ASKSTRING
    cell    field   askstring.row   \ row number
    cell    field   askstring.col   \ column number
    cell    field   askstring.text  \ address of text
    cell    field   askstring.len   \ length of text
    cell    field   askstring.ref   \ reference numb of column
end-struct
```

`ColListStyle: CLS-DRAWSELF`
The Column List style which determines how cell contents are drawn. Column lists with this style are sent CLM-DRAWCELL messages for each cell which should be drawn, with lParam set to the address of a /DRAWCOLLISTSTRUCT structure (defined below).

This style can be changed after the control has been created to alter the drawing behaviour accordingly.

A return value of 0 from this CLM-DRAWCELL is taken to indicate that the application did NOT draw the cell and the default drawing action is taken. This is usefull if an application wishes only to draw particular cells such as only text in a column. The /DRAWCOLLISTSTRUCT structure:

```
struct /DRAWCOLLISTSTRUCT
    int         dcls.Row        \ row number
    int         dcls.Col        \ column number
    int         dcls.Format     \ Windows DT_xxx format flags
    RECT field  dcls.ClipRect   \ Text clipping rectangle
```

```
    RECT field  dcls.CellRect   \ cell bounds
    int         dcls.text       \ address of text to print
    int         dcls.len        \ length of text to print
    int         dcls.hDC        \ handle of device context
    int         dcls.ref        \ reference number
    int         dcls.Selected   \ True if this row selected
end-struct
```

The address of a /CLDBLCLICK structure is passed as the lParam value of a CLM-DBLCLICKED message and is defined as follows:

The /CLDBLCLICK structure:

```
struct /CLDBLCLICK
    cell        field cldblclick.Handle     \ the handle of the column list
    cell        field cldblclick.ID         \ the ID of the column list
    cell        field cldblclick.row        \ the current row
    cell        field cldblclick.col        \ the current column
    cell        field cldblclick.xPos       \ horizontal mouse position
    cell        field cldblclick.yPos       \ vertical mouse position
    cell        field cldblclick.MouseRow   \ the row the mouse is pointing at
end-struct
```

`ColListStyle: CLS-ButtonHdr`
A column list style that causes the header control to have the HDS_BUTTONS style and be displayed as buttons. A column list with this style will send CLM-SORTCOLUMN messages when the user clicks on a header control button. This style has an effect during creation only - changing this style at run time does NOT alter the header style.

`ColListStyle: CLS-GRIDLINES`
A column list style specifying that the control is to show gridlines between columns and rows. This style can be changed after a column list has been created and the two words `cl-ShowGrid` and `cl-HideGrid` do just that.

`ColListStyle: CLS-InformSelf`
When a column list has this style specified, the messages CLM-ROWCHANGED and CLM-DBLCLICKED are sent to itself. Without this style messages are sent to the columnlist's parent window. This style may be changed after the control has been created to vary where messages are sent.

`ColListStyle: CLS-NoHdr`
Creates a column list with a header of zero height. The control then appears to have no header.

`ColListStyle: CLS-NoTrack`
Disables the resizing of columns using the mouse.

`ColListStyle: CLS-MultiSelect`
Allows more than one item to be selected by depressing the CONTROL or SHIFT keys while clicking on an item and/or pressing a cursor control key.

When a column list has this style, its *current row* need not be highlighted.

`ColListStyle: CLS-DragHdr`
Allows drag-and-drop of column header items. Only Column Lists with this file allow column dragging. If a column is dragged to a new position the columns are rearanged accordingly.

Notice that after a column is dragged to a new position, its index number is changed. Use the cl-Ref to identify columns in these cases.

Internals

`100 constant MaxCols`
The maximum number of columns allowed in a column list. A limit has to be defined since information about individual columns is held in fixed length arrays containing MaxCols items. The propertie of each column is held in an array of `ColProps` structures defined here:

```
struct /ColProps
    40 chars    field col.Title
    cell        field col.Width
    cell        field col.Align
    cell        field col.Ref
end-struct
```

`NextID: IDC_Hdr`
The ID of the header control which appears at the top of each column list.
Each Column List has an extended WinProps structure (see WINS.FTH and WINCTRLS.FTH), to hold information about the 'state' of the control. This is defined as follows:

The Column List window properties:

```
struct /CLProps
    /WinProps +
    cell        field   cl.cols         \ number of columns
    cell        field   cl.rows         \ number of rows
    cell        field   cl.Top          \ top row on display
    cell        field   cl.XOff         \ x-offset when horizontal scrolling
    cell        field   cl.HdrHeight    \ y-offset to avoid header
    cell        field   cl.MinHdrHt     \ minimum height
    cell        field   cl.LineHeight   \ height of row in pixels
    Cell        field   cl.CurrRow      \ row currently highlighted/selected
    Cell        field   cl.hHdr         \ handle of header control.
    cell        field   cl.hFont
    MaxCols /ColProps                   \ Array of columns properties
                Array-of cl.ColProps    \ (see ColProps above).
    cell        field   cl.MinColWidth  \ minimum column width
    RECT        field   cl.LineRect     \ holds line bounds during paint
    cell        field   cl.HScroll      \ true iff HSCroll showing
    cell        field   cl.VScroll      \ true iff VSCroll showing
    cell        field   cl.MultiSelect  \ true or false accordingly
    cell        field   cl.Selected     \ byte-array of rows selected
    cell        field   cl.Anchor       \ extended select anchor row
    cell        field   cl.Col          \ current colum/last column clicked
end-struct
```

Miscellaneous words used inside the control procedure

The following words are used ONLY inside the column list control procedure.

`: SetHeaderCols   \ --`
Sets the hdr column widths to match the column list.

`: clproc-l/p \ -- n ;`
Return the number of lines per page in the current column list. This number is floored (fractions are discarded from this positive number).

`: clproc-Max(l/p) \ -- n ;`
Return the ceiling of lines/page. That is any fractional part is rounded up.

`: clproc-SumWidths \ -- n`
Calculate the sum of all columns widths.

`: clproc-MinXOff \ -- n ;`
Return the most negative x-coordinate offset allowable. (The x-offset is the position of the left hand side of the left-most column). The value is used during horizontal scrolling operations.

`: clproc-TopMax \ -- n`
Return the maximum value which a top line can have. Used during vertical scrolling operations.

`: clproc-StretchRHCol \ --`
If the sum of all columns is less than the width of the window increase the width of right most colum so that the columns all fit in the control. If the sum of columns is greater than the window width no action is taken. If the resulting width would be less than the current width no action is taken.

`: clproc-LineSeen? \ line -- flag`
Given a line number, return TRUE iff that line is in view. Note that line is with reference to the display window. i.e line number zero is top most displayed line.

`: clproc-RowSeen? \ row# -- flag`
Given a row number, return TRUE iff that row can be seen in the window. Note the difference between row number and line number : A row number is the data index number. A line number is the line index within the current display window.

`: clproc-SpaceCols \ -- ;`
Make all columns same width ..

After use, this would best be followed by checking the HScroll bars - i.e. should they be shown? You can do this with the word `cl-ChkScrolls`.

`$80 $80 $80 Colour: GREY`
A constant COLORREF value. This colour is used to draw the vertical and horizontal lines that separates rows and columns.

`: GreyPen \ -- hPen`
Creates a sold 'pen' which is used to draw the 'grid lines'.

`: clproc-WM_Focus \ --`
WM_SETFOCUS and WM_KILLFOCUS handler. Simply re-paints the window.

Changing the rows displayed ...

`: clproc-MatchVScroll \ --`
Set the vertical scroll position to match the data being shown. The control is then repainted.

`: clproc-Top! \ n --`
Set the new top-most displayed line. n is restricted to be not less than zero and no more than `clproc-MaxTop`. `clproc-MatchVSCroll` is called.

`: clproc-Line+ \ n --`
Increase the top line number by n. Uses `clproc-Top!`

```
: clproc-Page+      \ n --
```
Adds n lots of `clproc-l/p` to the top line number which has the effect of 'scrolling' down n pages. If n is negative a scroll up occurs.

```
: clproc-WM_VScroll    \ --
```
The WM_VSCROLL message handler for the control. Performs the various up/down scrolling actions by calling `clproc-Line+` and `clproc-Page+` accordingly.

```
: clproc-Wheel  \ --    ;   \ Aug07
```
The WM_MOUSEWHEEL message. Simply does a 1-line scroll for each wheel rotation unit.

Responding to the WM_HSCROLL message

```
: clproc-XOffset!   \ n --
```
Set the current X-offset to n. n is a negative number representing the leftmost position of the left column. The value is non-zero when a horizontal scroll is in effect. n is limited to 'sensible' values, the header control is repositioned to match and the window is repainted.

```
: clproc-Left       \ n --
```
Shift the control left be n units (pixels). Uses `clproc-XOffset!`

```
: clproc-Right      \ n  -- ; move n chars right
```
Subtract n from the current X-offset value thus scrolling view right by n pixels.

```
: clproc-WM_HScroll    \ --
```
The WM_HSCROLL message handler for the column list control. Performs the horizontal scroll actions by calling the words `clproc-left` and `clproc-right` accordingly.

Controlling the vertical and horizontal scollbars

The following words control whether the vertical and horizontal scroll bars are displayed.

```
: clproc-VScroll?   \ -- flag
```
Return TRUE if a vertical scroll box is needed. A vertical scroll bar is needed when the number of rows in the list is greater than the number of lines which can be displayed.

```
: clproc-SetupVScroll   \ --
```
Set up/initialise the vertical scroll bar by setting the button size and the scroll range.

```
: clproc-VScroll+   \ --
```
Show and initialise the vertical scroll bar.

```
: clproc-VScroll-   \ --
```
Hides the vertical scroll bar. If the scrollbar is currently hidden, no action is taken.

```
: clproc-?VScroll   \ --
```
Hides or shows the vertical scroll bar as needed.

```
: clproc-HScroll?   \ -- flag
```
Returns true if horizontal scroll bar is needed. That is if the sum of column widths is greater than the window's client width.

```
: clproc-SetupHScroll   \ --
```
Setup/initialise the scroll horizontal bar by setting the scroll range and the scroll button size.

```
: clproc-HScroll+   \ --
```
Turn on (i.e. show) the horizontal scroll bar and set it up. If the scroll bar is already shown the it is set up only.

```
: clproc-HScroll-   \ --
```

Turn off (i.e. hide) the horizontal scroll bar. No action is taken if the scroll bar is not already shown.

`: clproc-?HScroll  \ --`
Shows the horizontal scroll bar if it is needed, otherwise the scroll bar is hidden.

Answering the WM_SIZE message

`: clproc-PosHdr  \ -- ; position header control`
Position the header control in accordance with the width and horizontal scroll position of the control.

`: clproc-WM_Size  \ -- ; WM_SIZE action`
The action taken in responce to a WM_SIZE message. Checks vertical and horizontal scroll states and sets the header controls size and position.

Handling left mouse button click messages

`: InformWho \ -- hWnd`
hWnd is the handle of the window to which CLM-ROWCHANGED and CLM-DBLCLICKED messages are to be sent. If the current window style includes the CLS-InformSelf bit, the current window handle is returned otherwise the current window's parent window is returned.

`: Send-RowChanged  \ --`
Send the CLM-ROWCHANGED message to the window specified by `InformWho`. The value of wParam is the current window handle, and lParam is the current row number.

`: clproc-EnsureSeen \ n --`
Ensure row number n is in view, altering the value of cl.top (the number of the top-most row shown) if neccessary.

`: clproc-Row!  \ n -- ;`
Set the current row to n. If n is less than zero the current row is set to zero. If n is greater than or equal to the number of rows the current row is set to the highest row number. If the current row is already at this value no action is taken. The topmost line is changed if neccessary to ensure the new current line is shown. Once the value has been set a CLM-ROWCHANGED message is sent to either the windows parent or to itself (depending upon the CLS-INFORMSELF style. See `SEND-ROWCHANGED`).

`: clproc-y>row# \ n1 -- n2`
Convert the vertical distance to a row number. The input parameter n1 is in pixels measured from the top of the window.

`: clproc-LClick(single) \ -- ;`
The WM_LBUTTONDOWN process when the control is in 'single selection' mode. The item over which the mouse points becomes the current item.

`: clproc-LClick(multi)  \ -- ;`
The WM_LBUTTONDOWN process. The item under the mouse pointer is selected unless the CONTROL or the SHIFT key is pressed. If the CONTROL key is pressed the item under the pointer is 'toggled' and if the SHIFT key is pressed the items between that and the current item are all selected. In each of these cases the current item (i.e. cl.CurrRow value) does not change. If neither the CONTROL key or the SHIFT key is pressed the item under the pointer is selected and all other items are deselected.

`: clproc-LClick  \ -- ;`
The WM_LBUTTONDOWN process. Does either clproc-LClick(single) or clproc-LClick(multi) depending on the value of cl.MultiSelect property.

```
: clproc-LDblClick      \ -- ;
```
The WM_LBUTTONDBLCLK message action. Send the CLM-DBLCLICKED message to either column list itself or its parent window depending on whether the column list has the CLS-InformSelf style. For the CLM-DBLCLICKED message wParamLo is the ID of the column list and lParam is the address of a /CLDBLCLICK structure. wParamHi is not used.

Initialisation

```
: clproc-InitWidths     \ --
```
Make all columns 4 characters wide. Used when the control is being initialised.

```
: clproc-ChkHdrBtns     \ --
```
Called during initialisation and by the WM_STYLECHANGED message handler (involked when a window style is changed). If the column list has the CLS-ButtonHdr style the header control is given the HDS_BUTTONS style, if not HDS_BUTTONS is removed. Similarly the HDS_DRAGDROP style is added to or removed from the header control according to whether the column list control has the HDS_DRAGDROP style.

```
: clproc-SetHdrHeight   \ --
```
Called during creation and when the control's style changed by the WM_STYLECHANGED handler. Sets the header control's height according to the value of the CLS-NoHdr style.

```
: clproc-WM_Create  \ --
```
WM_CREATE message handler. Performs a number of tasks to initialise the control.

Header Notification message handling

The header control sends WM_NOTIFY messages to its parent (i.e. the column list) whenever it is clicked or a section divider is dragged. The following words provide actions to respond to these events.

```
: clproc-HdrTrack   \ -- ;
```
The word executed in response to the HDN_ENDTRACK notification message. The header section width is set to the new indicated position and the column width is set to match. The horizontal scroll requirements are checked and the control is finally re-painted.

```
: clproc-HdrClick   \ --
```
The action performed in response to a HDN_ITEMCLICK notification message. The header must have the HDS_BUTTONS style to generate this message. The single action performed is to send a CLM-SORTCOLUMN message to the parent window if HDN_ITEMCLICK was caused by a left button click. If a non-zero result is returned the window is re-painted.

```
: clproc-Notify-Hdr \ --
```
The WM_NOTIFY message handler for messages sent by the header control.

```
: clproc-WM_Notify      \ -- ; WM_NOTIFY handler
```
The WM_NOTIFY message handler.

Handling key down events

```
: clproc-GetDlgCode     \ --
```
The action performed in response to a WM_GETDLGCODE message. This word supplies the default value (obtained by calling the default control procudure) ORed with DLGC_WANTARROWS, thus informing Windows that this control is to receive WM_KEYDOWN messages whenever a cursor control key is pressed.

```
: clproc-SingleRow+     \ n --
```

Add n to the current row number and deselects the previously selected items. This is a factor of `clproc-WM-KeyDown` - the WM_KEYDOWN message hadler. Calls `clproc-SingleRow!`

`: clproc-WM_KeyDown(single)  \ --`
The WM_KEYDOWN message handler when in single-select mode. The routine detects the cursor control keys (e.g. UP, DOWN, PAGE DOWN etc.) and adjusts the current row accordinly. Uses `clproc-SigleRow+` and `clproc-SigleRow!`.

`: clproc-ExtendRow+  \ n --`
A factor of `clproc-WM_KeyDown(multi)`, the key down message handler when the control is acting in its multi-select mode. n is added to the current row to become the new current row and all rows between that row and the anchor row are selected.

`: clpKeyStates  \ -- u ;`
Returns a set of bit flags representing the state of the SHIFT, CONTROL and ALT keys. Bit $01 indicates the ALT key is pressed, $02 indicates a SHIFT key is pressed and $04 indicates the CONTROL key is pressed.

`: clproc-CTRLKey  \ key --`
A factor of clproc-WM_KeyDown(multi) executed if the CTL key is depressed. Depending on the key press

`: clproc-WM_KeyDown(multi)  \ --`
The WM_KEYDOWN message handler. The routine calls `clproc-SHIFTKey` if the SHIFT key is down, `clproc-CTRLKey` if the CONTRLOL key is down but otherwise calls `clproc-WM_KeyDown(single)`.

`: clproc-WM_KeyDown  \ --`
The WM_KEYDOWN message handler. Performs either `clproc-WM_KeyDown(single)` or `clproc-WM_KeyDown(multi)` depending on the value of `cl.multiselect`.

Getting and Setting the font

By default the action of creating a control causes the font of the control to be matched to its parent. (See `MatchFont` in *Wintls.fth*). For a control containing other controls however, further action is necessary in order to match all the control's fonts.

The two words below provide action for both the WM_SETFONT and WM_GETFONT messages and ensure that the font used in the header control is the same as the main column list control.

`: clproc-WM_SetFont \ --`
The WM_SETFONT message handler. The font handle passed as wParam is saved in the current window properties and the header font is set to that font.

`: clproc-WM_GetFont  \ --`
The WM_GETFONT message handler. Returns the font handle as held in the control's window properties.

WM_Paint handler words

`: clproc-CellStr  \ col# row# -- caddr ulen`
Provides the address and length of the text string which is to occupy the cell at column *col#* on row *row#*, by sending a CLM-ASKSTRING message. If the column list has the CLS-ASKSELF style the message is sent to itself otherwise the message is sent to its parent window.

`: clproc-LineHt  \ --n`

Returns the current line height.

`: clproc-LineTop \ line# -- y`
Returns position of the top of the given line number.

`: clproc-SetLineRect \ line# --`
Calculate the display line coordinates and fill the current columnlist winprops cl.LineRect field.

`: clproc-ColLeft \ col# -- left`
Calculate the left position of the given column.

`: clproc-ColWidth \ col# -- width`
Returne the width of column number *col#*.

`: clproc-CellBounds \ col# line# -- left top right bottom`
Return the coodinates in pixels of the cell at line *line#* column number *col#*.

`: clproc-CellClip \ col# line# -- left top right bottom`
Return the coodinates in pixels of the clipping rectangle for the cell at line *line#* column number *col#*. The clipping rectangle is that area outside of which text is not shown.

`: clproc-VLine \ y x --`
Paint (Draw?) a vertical line from the point (x,0) to (x,y). The word is a factor of `clproc-VLines`.

`: clproc-VLines \ --`
Draws the vertical lines between 'cells' in the client area of the control.

`: clproc-HLine \ xmax y --`
Draw a horizontal line from (0,y) to (xmax,y) in the column list window. A factor of `clproc-HLines`.

`: clproc-HLines \ --`
Draws the horizontal lines between lines in the client area of the control.

`: clproc-GLines \ --`
Draw the gridlines for the current control. No action is taken if the window style CLS-GridLines bit is not set.

`: clproc-FmtFlags \ col# -- flags`
Returns the format bit-flags for columns number *col#*.

`: clproc-FillLine \ hBrush --`
A factor of `clproc-BlankLine` and `clproc-HiLightRow`, which are part of the window painting routine. Fills the current line rectangle with the given brush.

`: clproc-BlankLine \ --`
Blanks the current line rectangle.

`: clproc-WriteCell \ row# line# col# --`
Draws the text for data row *row#* column number *col#* at line *line#* in the current window This is the cell-drawing primitive.

`: clproc-SelfDraw? \ -- flag`
Return true iff the current window style bits include CLS-DRAWSELF.

`: clproc-DrawText \ --`
For use within a CLM-DRAWCELL handler by a child of this control, this word draws text in a format defined within the /DRAWCOLLISTSTRUCT structure held as lParam. The word

is EXPORTED in order to be used by children of this control since its action will be the most common one.

```
: clproc-FillRect       \ colour --
```

To be used as part of CLM-DRAWCELL handler this word is EXPORTed. Fills the cell rectangle with the COLORREF value supplied. The cell rectangle is part of the /DRAWCOLLISTSTRUCT structure held as lParam.

```
: clproc-SendDrawCell   \ row# line# col# -- n
```

Send a CLS-DRAWCELL message to the current window returning the value returned by SendMessage. A /DRAWCOLLISTSTRUCT is prepared and sent with the message.

```
: clproc-DrawSelfCell   \ row# line# col# --
```

Sends a CLM-DRAWCELL message to the current window (i.e. itself) by calling `clproc-SendDrawCell`. If the returned value is zero (indicating that the drawing did not take place) `clproc-WriteCell` is executed. In this way a developing application is more likely to see some text before the CLM-DRAWCELL message handler is completed. On the other hand the text drawn by a message handler that returns zero in error, will see its own efforts overwritten!

```
: clproc-DrawSelfRow    \ row# line# --
```

Draws/Paints the given row at the given line number by repeatedly calling `clproc-DrawSelfCell`. This word is executed if the control has the **CSL-DrawSelf** window style.

```
: (clproc-WriteRow) \ row# line# --
```

A factor of `clproc-WriteRow`. Draws a line of data by repeatedly calling `clproc-WriteCell`.

```
: clproc-WriteRow   \ row# line# --
```

Paint the given row number at the display line position given. row is known to be a valid row number. The current background brush and the pen is not changed. Either `clproc-DrawSelfRow` of `(clproc-WriteRow)` is called depending on whether the control has the **CSL-DrawSelf** window style.

```
: clproc-HiLightRow     \ row# line# --
```

Highlights the given row/line by setting the text and background colours COLOR_HIGHLIGHTTEXT and COLOR_HIGHLIGHT respectively and calling `clproc-WriteRow`. Previous colours are restored on leaving this word.

```
: clproc-OutlineRect    \ --
```

Draw a 'focus rectangle' around the row being currently drawn

```
: (clproc-paintrow) \ row# line# --
```

Paint the given row number at the display line position given. row is known to be a valid row number. Calls `clproc-WriteRow` or `clproc-HitlightRow` if row# is the current row number.

```
: clproc-PaintRow   \ row# line# --
```

Paint the given row of data at display line given. Uses `(clproc-paintrow)`.

```
: clproc-PaintLines \ --
```

Paints those lines which are being displayed. A factor of `clproc-WM_Paint`.

```
: clproc-WM_Paint       \ -- ;
```

The handler for the WM_PAINT message. Calls `clproc-PaintLines` and `clproc-GLines`

Handling the *set property* messages

```
: clproc-SetCols    \ --
```

The **CLM-SetProps** message handler for the *Set number of columns* command. The number of columns is set to the value wParam. The view is set to be 'scrolled' fully to the left. The header sections is matched to the columns.

`: clproc-SetRows \ --`

The **CLM-SetProps** message handler for the *Set Number of Rows* command. Sets the number of rows to the value wParam, sets the top row to zero and adjusts the vertical scroll bars as necessary. No action is taken if there is no change to the number of rows.

`: clproc-SetColWidth \ --`

The **CLM-SetProps** message handler for the *Set column width* command. For this command wParamLo is column number, wParamHi is required width.

`: clproc-Clear`

The **CLM-SetProps** message handler for the *Clear* command. The number of rows is set to zero, the current row is set to -1 and the window repainted.

`: clproc-EqualWidths`

The **CLM-SetProps** message handler for the *Equalise column widths* command. Makes all columns of equal width. (The rightmost column will be slightly wider than the others as a result of 'raound off' errors.

`: clproc-SpreadCols \ --`

The **CLM-SetProps** message handler for the *i{Spread Columns} command. Adjusts column widths so they cover the control's width.

`: clproc-SetCurrRow \ --`

The **CLM-SetProps** message handler for the *Set the current row* command. Sets the current row to wParam and adjusts the display to ensure that row is visible. If the current row is thus changed a CLS-ROWChanged message is sent.

`: clproc-SelectRow \ --`

The action taken for a **CLM-SetProps** function number 6. wParamLo is a row number which is marked as 'selected'. The control is then marked as 'invalid' to cause a repaint.

`: clproc-DeSelectRow \ --`

The action taken for a **CLM-SetProps** function number 7. wParamLo is a row number which is marked as 'unselected'. The control is then marked as 'invalid' to cause a repaint.

`: clproc-SetLineHeight \ --`

The action taken for a **CLM-SetProps** function number 10 - Set Line Height. wParamLo is the height required in pixels.

`: clproc-CLM-SETPROP`

The handler for the CLM-SETPROP message which is used to set various properties:

`: clproc-RowSelected? \ -- flag`

The action taken in response to a CLM-GETPROP message function 6. True is returned if the row number held in wParamLo is marked as selected, otherwise FALSE is returned.
• lParamLo = 13 : Used to get string for column wParamlo, row wParamHi

`: clproc-CLM-GETPROP \ --`

The handler for the CLM-GETPROP message which is used to get various properties:

`: clproc-CLM-CHK \ --`

CLM-CHKControl message handler.

The ColumnList Class and the ColumnList Control

The column list control and the window class on which it is based are defined below.

```
Winclass: ColumnListClass
    /CLProps PropertySize

\    Name "GUW - Column List WINDOW"

    CS_STANDARD Style
\    Color_Window Brush-colour
    0 Menu
\    ClassProc CL-ClassProc

    0 0 100 20 IDC_Hdr      z" Header"      cl-HeaderControl

    WM_STYLECHANGED     does clproc-WM_STYLECHANGED
    WM_CREATE           does clproc-WM_CREATE
    WM_PAINT            does clproc-WM_paint
    WM_LBUTTONDOWN      does clproc-LClick
    WM_LBUTTONDBLCLK    does clproc-LDblClick
\    WM_RBUTTONDOWN      does clproc-LClick
    WM_SIZE             does clproc-WM_Size
    WM_VSCROLL          does clproc-WM_VScroll
    WM_HSCROLL          does clproc-WM_HScroll
    WM_NOTIFY           does clproc-WM_Notify
    WM_SETFOCUS         does clproc-WM_Focus
    WM_KILLFOCUS        does clproc-WM_Focus
    WM_KEYDOWN          does clproc-WM_KeyDown
    WM_GETDLGCODE       does clproc-GetDlgCode
    WM_SETFONT          does clproc-WM_SetFont
    WM_GETFONT          does clproc-WM_GetFont
    CLM-SETPROP         does clproc-CLM-SetProp
    CLM-GETPROP         does clproc-CLM-GetProp
    CLM-CHKControl      does clproc-CLM-CHK
    WM_MOUSEWHEEL       does clproc-Wheel           \ Aug07
\ 4Oct11    WM_ENABLE           does clproc-Enable

end-WinClass

ColumnListClass AppControl: ColumnList
\ 27Apr13    WS_CONTROL WS_BORDER or Style
    WS_CONTROL              Style
    WS_VSCROLL              Style+
    WS_TABSTOP              Style+
    WS_EX_CLIENTEDGE        ExStyle
    clr-Window Colour
end-control
```

Application words to control the ColumnList

The following words set and retreive the various properties of a column list. Each word is prefixed by *CL-* to indicate 'ColumnList'.

```
: cl-SetCols    \ #cols hCtl --
```
Set the number of columns to #cols.

```
: cl-SetRows    \ #rows hCtl --
```
Set the total number of rows the #rows.

```
: cl-SetColWidth    \ width hCtl col# --
```
Set the width of the given column number to 'width'.

```
: cl-Clear  \ hCtl ---
```
Clear or empty the column list of all data. This is accomplished by setting the number of rows to zero.

```
: cl-EqualWidths    \ hCtl --
```
Makes each column of eaqual width covering the whole of the window.

```
: cl-SetCurrRow \ n hCtl --
```
Sets the current row number to n and moves that row onto the display if not already seen.

```
: cl-Cols       \ hCtl -- n
```
Returns the number of columns.

```
: cl-Rows       \ hCtl -- n
```
Return the total number of rows.

```
: cl-ColWidth   \ hCtl col# -- width
```
Return the width of the specified column.

```
: cl-LineHeight     \ hCtl -- n
```
Returns the height is pixels of a line.

```
: cl-SetLineHeight  \ n hCtl --
```
Sets the line height to n pixels.

```
: cl-Top        \ hCtl -- n
```
Returns the top line number. That is the index number of the line which is displayed topmost within the window.

```
: cl-SetTop     \ n hCtl --
```
Set the line index of the line displayed at the top of the control.

```
: cl-ShowGrid   \ hCtl --
```
Causes grid lines to be dislayed by adding the CLS-GridLines window style and forcing the window to be redrawn.

```
: cl-HideGrid   \ hCtl --
```
Removes grid lines by taking the CLS-GridLines window style from the column list.

```
: cl-CurrRow    \ hCtl -- row#
```
Return the current row number.

```
: cl-ButtonHdr  \ hCtl --
```
Makes the header appear as a button header.

```
: cl-StandardHdr    \ hCtl --
```

Makes the header appear as a standard header rather than a button header.

`: cl-Alignment  \ hCtl col# -- n`
Return the formatting code for the specified column.

`: cl-SetAlign   \ align# hCtl col# --`
Set the text alignment value for the specified column. Note that the Windows value DT_NOPREFIX (i.e. 2048) should be added (ORed) to the value of align# to stop an ampersand character ('&') being interpreted as a directive to underscore the character that follows.

`: CL-RALIGN \ hCtl col# --`
Set that text within the specified column should be 'right justified'.

`: CL-LALIGN \ hCtl col# --`
Set that text within the specified column should be 'left justified'.

`: CL-CENTRED    \ hCtl col# --`
Set that text within the specified column should be 'centre justified'.

`: cl-Title      \ hCtl col# -- caddr ulen`
Returns the title text for the given column.

`: cl-SetTitle   \ c-addr len hCtl n --`
Sets the title for the given column.

`: cl-Title      \ hCtl col# -- caddr ulen`
Returns the title text for the given column.

`: cl-ChkScrolls     \ hCtl --`
Performs a check on the scroll state of the control. The check performed is dependant on the value of wParamLo: wParamlo = 0 : Check visibility of horizontal scroll. wParamlo = 1 : Check visibility of vertical scroll. wParamlo = 2 : Check visibility of both scroll bars.

`: cl-ChkCols        \ hCtl --`
Checks that the header control is positioned correctly and that the sum of the columns is at least as wide as the control's client width by stretching the right-most column if necessary. This is done by posting a CLM-CHKControl message to the control.

`: cl-ChkHdr         \ hCtl --`
Checks that the header control is positioned and sized correctly. This is done by posting a CLM-CHKControl message to the control.

`: cl-Ref        \ hCtl col# -- addr`
Returns the address of a column's reference number - A 32-bit value for use by the application.

`: cl-AddCol     \ caddr ulen width ref hCtl --`
Create a new column with a title given by the text string *caddr/ulen*, of width *width* horizontal units (see `hUnits`) and with the reference number *ref*.

14.2.1 Initialising the columns

The words `cl-InitCol` and `cl-InitCols` below setup column properties supplied by a application in a table as described below.

The InitCols table consists if a single cell containing the

- Column title as counted string.
- Column width as a single cell, and

• Column alignment, as a single cell.

Not that each field starts on al `aligned` address and that the title field length is variable. The second field is reached from the first by the phrase "count + aligned". An example table is shown below:

```
Create MDlgCols
```

```
: cl-InitCols   \ table hCtl --
```
Initialises a column list as specified in the given table.

```
: cl-SetWidths  \ table hCtl --
```
Sets the column widths as defined in the initialisation table given.

14.3 Application Words

The following list of exported words are those used by applications which contain column list controls.

Stack Notation

- *n*,*n1* and *n2* are integers
- *#cols* and *#rows* are integers indicating a number of colums and a number of rows respectively.
- *col#* and *row#* are integers indicating a column number and a row number respectively.
- *width* is an integer idicating a width.
- *hCtl* is the Windows handle of a column list control.
- *c-addr* is a character aligned address and *len* is a length and when used together indicate a text string of *len* characters.

Column List Style Constants

```
Export  CLM-ROWCHANGED
Export  CLM-DBLCLICKED
Export  CLM-SORTCOLUMN
Export  CLM-ASKSTRING
Export  CLM-DRAWCELL
Export  CLS-ASKSELF
Export  CLS-DRAWSELF
Export  CLS-ButtonHdr
Export  CLS-GRIDLINES
Export  CLS-InformSelf
Export  CLS-NoHdr
Export  CLS-NoTrack
Export  CLS-MultiSelect
Export  CLS-DragHdr
```

Data Stuctures

```
Export  /ASKSTRING      \ -- n
Export  askstring.row   \ n1 -- n2
Export  askstring.col   \ n1 -- n2
Export  askstring.text  \ n1 -- n2
Export  askstring.len   \ n1 -- n2
```

```
Export  askstring.Ref   \ n1 -- n2

Export  /DRAWCOLLISTSTRUCT  \ -- n
Export  dcls.Row        \ n1 -- n2
Export  dcls.Col        \ n1 -- n2
Export  dcls.Format     \ n1 -- n2
Export  dcls.ClipRect   \ n1 -- n2
Export  dcls.CellRect   \ n1 -- n2
Export  dcls.text       \ n1 -- n2
Export  dcls.len        \ n1 -- n2
Export  dcls.hDC        \ n1 -- n2
Export  dcls.Selected   \ n1 -- n2

Export  /CLDBLCLICK         \ -- n
Export  cldblclick.Handle   \ n1 -- n2
Export  cldblclick.ID       \ n1 -- n2
Export  cldblclick.row      \ n1 -- n2
Export  cldblclick.col      \ n1 -- n2
Export  cldblclick.xPos     \ n1 -- n2
Export  cldblclick.yPos     \ n1 -- n2
Export  cldblclick.MouseRow \ n1 -- n2
```

Control procedure words

```
Export  clproc-DrawText     \ --
Export  clproc-FillRect     \ colour --
```

Column list manipulation words

```
Export  cl-SetCols          \ #cols hCtl --
Export  cl-SetRows          \ #rows hCtl --
Export  cl-SetColWidth      \ width hCtl col# --
Export  cl-Clear            \ hCtl ---
Export  cl-EqualWidths      \ hCtl --
Export  cl-SetCurrRow       \ n hCtl --
Export  cl-Cols             \ hCtl -- n
Export  cl-Rows             \ hCtl -- n
Export  cl-ColWidth         \ hCtl col# -- width
Export  cl-LineHeight       \ hCtl -- n
Export  cl-SetLineHeight    \ n hCtl --
Export  cl-Top              \ hCtl -- n
Export  cl-SetTop           \ n hCtl -- n
Export  cl-ShowGrid         \ hCtl --
Export  cl-HideGrid         \ hCtl --
Export  cl-CurrRow          \ hCtl -- row#
Export  cl-ButtonHdr        \ hCtl --
Export  cl-StandardHdr      \ hCtl --
Export  cl-Alignment        \ hCtl col# -- n
Export  cl-SetAlign         \ align# hCtl col# --
Export  cl-RALIGN           \ hCtl col# --
```

```
Export  cl-LALIGN           \ hCtl col# --
Export  cl-CENTRED          \ hCtl col# --
Export  cl-Title            \ hCtl col# -- caddr ulen
Export  cl-SetTitle         \ c-addr len hCtl n --
Export  cl-ChkScrolls       \ hCtl --
Export  cl-ChkCols          \ hCtl --
Export  cl-ChkHdr           \ hCtl --
Export  ColumnList          \ --
Export  cl-Ref              \ hCtl col# -- addr
Export  cl-AddCol           \ caddr ulen width hCtl --
Export  cl-Select           \ hCtl row# --
Export  cl-DeSelect         \ hCtl row# --
Export  cl-SetSelect        \ flag hCtl row# --
Export  cl-SelectAll        \ hCtl --
Export  cl-SelectNone       \ hCtl --
Export  cl-Selected?        \ hCtl row# -- flag
Export  cl-SelectCount      \ hCtl -- n
Export  cl-SpreadCols       \ hCtl --
Export  cl-TotColWidth      \ hCtl -- n
Export  cl-y>Row            \ y hCtl -- row#
Export  cl-x>Col            \ x hCtl -- col#
Export  cl-InitCols         \ table hCtl --
Export  cl-SetWidths        \ table hCtl --
Export  cl-Hdr              \ hCtl -- hHdr
Export  cl-CellStr          \ col# row# hCtl -- caddr ulen
Export  ColumnListHeaderID  \ -- n
Export  cl-ColumnMode       \ hCtl --
Export  cl-RowMode          \ hCtl --
Export  cl-ColumnMode?      \ hCtl -- flag
Export  cl-SetColumn        \ n hCtl --
Export  cl-GetColumn        \ hCtl -- n
Export  cl-CellRect         \ hCtl -- x y w h
Export  cl-SetMinColWidth   \ n hCtl --
Export  cl-MinColWidth      \ hCtl -- n
Export  cl-SetMinHdrHt    \ n hCtl --
Export  cl-MinHdrHt       \ hCtl -- n
```

14.4 Example

In the spirit of object orientated programming, a control will be expected to draw itself. To do this we will create a child column list giving it the CLS-DRAWSELF. This causes the control to receive a CLM-DRAWCELL message whenever a cell value is to be drawn. The CLM-DRAWCELL message supplies information including the row and column numbers concerned and the coordinates in which the text should be drawn.

The text string to draw is supplied by the CLM-DRAWCELL message having sent a CLM-ASKSTRING message beforehand. By default the parent window is asked this question, but our sub-classed control will be asked instead since it has the CLS-ASKSELF style.

The word `CellText` supplies the text string for a cell in answer to the CLM-ASKSTRING. In this example the column and row numbers are returned as text.

```
: CellText  \ --
    \ Prepare the string...
    <#  lParam askstring.row @ 0 #s 2drop
        [char] , hold
        lParam askstring.col @ 0 #s #>  \ caddr ulen
    \ store it in the structure whose address is held in lParam...
    lParam askstring.len !
    lParam askstring.text !
    \ All done!
;
```

Drawing a cell's contents is done by `DrawCell`. It has a single factored word `BkdgCell` which draws the background colour of the cell. If this is the currently selected row the main column list painting routine has already painted it with the colour COLOR_HIGHLIGHT, so no action is taken in that case. Otherwise a background colour of either red or cyan depending on the value of the current row, is painted.

```
: BkdgCell  \ --
    \ If this is the current row, no action is taken,
    \ otherwise the background is to be either red or cyan...
    lParam dcls.Row @ dup CurrWin cl-CurrRow =
    if  drop
    else  2 mod if Red else Cyan then clproc-FillRect
    then
;
```

After filling in the background colour, `DrawCell` draws the text supplied to it. Before doing so it sets the background mode to transparent, saving the previous mode on the return stack. The text is drawn using the EXPORTed word `clproc-DrawText` after which the background mode is restored.

In order to tell the base control painting routine that the cell has indeed been drawn a non-zero value is returned. `DO-DEFWNDPROC?` if turned off to prevent the default window procedure being performed which causes the return value to become zero.

```
: DrawCell  \ -- ; wParam=? , lParam=/DRAWCOLLISTSTRUCT struct
    BkdgCell
    lParam dcls.hDC @ TRANSPARENT SetBkMode >r
    clproc-DrawText
    lParam dcls.hDC @ r> SetBkMode drop
    1 ReturnVal !
   -inherited
;
```

Our column list will need setting up. It needs to "know" how many rows and columns it has and what the column titles are.

```
Create CLCols
    8 ,
    ", Centred"     10 ,    DT_CENTER DT_NOPREFIX or ,
    ", Default"     10 ,    DT_LEFT DT_NOPREFIX or ,
    ", RAligned"    10 ,    DT_RIGHT DT_NOPREFIX or ,
    ", three"       10 ,    DT_LEFT DT_NOPREFIX or ,
    ", four"        10 ,    DT_LEFT DT_NOPREFIX or ,
    ", five"        10 ,    DT_LEFT DT_NOPREFIX or ,
    ", six"         10 ,    DT_LEFT DT_NOPREFIX or ,
    ", seven"       10 ,    DT_LEFT DT_NOPREFIX or ,
```

Initialising the column list

```
: init-MyColList
    CLCols CurrWin cl-initcols
    100 CurrWin cl-SetRows      \ and 100 rows.
    CurrWin cl-EqualWidths
;
```

The definition of our control can now be made. It is called `NewColumnList` and has three message handlers assigned to it.

```
another ColumnList NewColumnList
    CLS-ASKSELF Style+
    CLS-DRAWSELF Style+
    CLS-GridLines Style+
    CLS-DragHdr Style+

    CLM-ASKSTRING   does CellText
    CLM-DRAWCELL    does DrawCell
    GWM_InitControl does init-MyColList

(( WM_NOTIFY does:
    case lParam NMHDR.Code @
    HDN_BEGINTRACK          of  ." Begin"   cr  endof
    HDN_DIVIDERDBLCLICK     of  ." DIV CLICK"   cr  endof
    HDN_ENDTRACK            of  ." ENDTRACK"    cr  endof
    HDN_ITEMCHANGED         of  ." Changed" cr       endof
    HDN_ITEMCHANGING        of  ." CHANGING"    cr  endof
    HDN_ITEMCLICK           of  ." Click" cr     endof
    HDN_TRACK               of  ." Track"     endof
                                ." SUMAT ELSE" cr
    endcase
;
))

end-control
```

Now that we have our control, we need a window to put it in. Below is defined a dialog box which contains a button control labelled "Close", and our column list control `NewColumnList`.

Additionally controls have been added which allow us to demonstrate the some of the CLM-xxx messages.

Windows requires that each control has a unique identifier. The following words provide some:

```
NextID: IDC_ColList     \ ID for our column list
NextID: IDC_CLOSE       \ ID for the button to close the dialog box
NextID: IDC_CurrRow     \ for label showing Current Row
NextID: IDC_LastRow     \ for label showing last selected row
```

The dialog box procedure is defined next. It handles messages from the *Close button* (in the form of a WM_COMMAND) and a number of messages sent by the column list control. CLM-DBLCLICKED and CLM-RowChanged are detected to display the number of the currently highlighted row and the the row number that was last 'selected'.

Perhaps puzzling is the presence of a message handler for CLM-ASKSTRING. Our column list has been given the style CLS-ASKSELF and so this dialog procedure will never detect that message. As an experiment try removing the CLS -ASKSELF from our columnlist.

Now the dialog box is defined.

```
Dialog: TestDlg

    10 10 250 130 Position&Size

    WS_VISIBLE WS_CAPTION or WS_SYSMENU or WS_THICKFRAME or Style
    DS_MODALFRAME DS_3DLOOK or DS_SetFont or Style+

    10 100 0 Font "MS Sans Serif"

    2 2 190 95    IDC_ColList   z" Column list "        NewColumnList
    205 20 40 15    IDC_CLOSE   z" &Close"              DefPushButton
    2 100 60 24    IDC_STATIC   z" Selected Row"         GroupBox
    20 110 40 10   IDC_LastRow  z" -"                   TextLabel
    70 100 80 24   IDC_STATIC   z" Current Row"         GroupBox
    80 110 40 10   IDC_CurrRow  z" -"                   TextLabel

end-dialogbox
```

To test this code, execute the word *CLTest* defined below. The word ensures the dialog box is modeless so that the Forth console will continue to operate. The value returned from a modeless dialog box is its handle and this is set as the *current window*.

```
: CLtest   \ --
    0 TestDlg ShowModeless Set-CurrWin
;
```

15 Grid Control

15.1 Introduction

The Grid Control is a custom window control used to display data as a "grid" in a similar way to a spread sheet. Unlike a Column List control which tends to handle data on a line by line basis, a Grid Control views data as individual 'cells'. (A cell in this sense is that unique location at the intersection of a row and column).

Similar to the Column List however, the grid is topped by a header control which contains column headings and which allows column sizing by dragging header section dividers. If the control has the `GCS-ButtonHdr` style a `GCM_SortColumm` message is generated whenever a header section is clicked.

Each cell's data is assumed to be text and is supplied by the application responding to a `GCM-AskString` message. If the control has the `GCS-TalkSelf` style the control itself receives these messages, otherwise the control's parent will receive them.

A capability which can be particularly useful is the 'cell editor' where the text of an individual cell can be edited by double-clicking in a cell. By answering the `GCM-QueryEdit` message accordingly, an application can choose to let the cell be edited or not, or to provide its own editing function.

A number of 'character filters' may be chosen to control the type of data which may be entered into a cell.

Public data structures, control messages and control styles

The string values of grid cells are supplied by applications when they answer `GCM-AskString` messages. The grid control supplies row and column numbers and receives an address and length of text. Background and text colour values may also be sent which are then used when drawing the text on screen.

The /GCStrspec structure is defined as follows:

```
    struct /GCStrspec
        cell    field   GCStrspec.row           \ row number
        cell    field   GCStrspec.col           \ column number
        cell    field   GCStrspec.text          \ address of text
        cell    field   GCStrspec.len           \ length of text
        cell    field   GCStrspec.Align         \ alignment
        cell    field   GCStrspec.Coloured      \ flag ; true if coloured
        cell    field   GCStrspec.TextColour    \ COLRORREF for text
        cell    field   GCStrspec.BkgdColour    \ COLRORREF for Bkgd
        cell    field   GCStrspec.Ref           \ copy of gc.Ref value
\ 20Feb09        cell    field   GCStrspec.TermKey       \ key code which terminated edit
    end-struct
```

the gcTermEdit structure is used to supply information to an application when a grid cell has been edited. It is an extension of /GCStrspec above:

```
struct /gcTermEdit
    /GCStrspec +
    cell    field   termedit.TermKey    \ key code which terminated edit
    cell    field   termedit.CtrlDown   \ TRUE if the Control key is pressed
    cell    field   termedit.ShiftDown  \ TRUE if the Shift key is pressed
    1 kb    field   termedit.text       \ text of edit when terminated
end-struct
```

Whenever a key is pressed while editing a grid cell, a GCM-AskTermEdit message is sent to the Grid control's parent (or to the Grid Control if it has the GCS-TALKSELF style). A GCM-AskTermEdit message supplies data about the key press in the following data structure:

```
struct /AskTermEdit
    cell    field   gcate.TermKey   \ key code of key pressed
    cell    field   gcate.CtrlDown  \ TRUE if the Control key is pressed
    cell    field   gcate.ShiftDown \ TRUE if the the Shift key is pressed
    1 kb    field   gcate.text      \ text of edit when terminated
end-struct
```

If a grid control has the `GCS-DrawSelf` style, `GCM-DrawCell` messages are sent (to itself) when a cell is to be drawn. The /DrawGridCell structure is then used to supply enough information for the subclassed control to draw the cell.

The /DRAWGRIDCELL structure is defined below.

```
struct /DRAWGRIDCELL
    RECT        field  dgc.ClipRect \ the rectangle text is confined to
    RECT        field  dgc.CellRect \ the cell bounding rectangle
    int                dgc.hDC      \ handle of device contect
    /GCStrspec  field  dgc.GCss     \ a string spec structure.
end-struct
```

An application uses the /DRAWGRIDCELL structure to draw a cell in the following way.

1. dgc.GCss is inspected and if its GCStrspec.Coloured field is set to true then the value of GCStrspec.TextColour is a COLORREF value used to fill the area bounded by dgc.CellRect. The text is drawn using that colour in the GCStrspec.BkgdColour field.
2. If GCStrspec.Coloured is false text is drawn without specifying a colour.

In both cases GCStrspec.Align holds a Windows DT_xxx value as used in the Windows function DrawText. The following procedures are 'used internally' by the grid control but are `export`ed to be available to applications. Each word consumes a single DRAWGRIDCELL parameter from the stack and uses values held in it to perform some act.

- `dgc-ColourText` draws coloured text.
- `dgc-ColourCell` fills the cell with a colour.
- `dgc-DrawText` simply draws the text.
- `dgc-PaintCell` makes the decision of whether to use the colour words or not.

The GCM-QueryEdit message (see later) is sent when a grid control wishes to know if a particular

cell may be edited. To do so it sends a /GCCellSpec structure which the application modifies to provide editing parameters.

As of December 2008, a new message `GCM-CharEdit?` is sent whenever a key is pressed and this structure is sent with it. The field gccp.Char is used only with this new message.

The /GCCellSpec structure is defined as follows:

```
struct /GCCellSpec
    int                 gccp.Col        \ current column number
    int                 gccp.Row        \ current row number
    RECT field          gccp.CellBounds \ bounding coordinates of cell
    int                 gccp.CanEdit    \ return true if editing allowed.
    int                 gccp.DataType   \ See GCDT-xxx constants below.
    int                 gccp.MaxChars   \ 0 or maximun no. of chars.
    int                 gccp.ref        \ copy of gc.ref value
    int                 gccp.Char       \ char code of key pressed
end-struct
```

The /AskSelect data structure is used to pass data to an application using the GWM-AskSelect message. See GWM-AskSelect later.

```
struct /AskSelect
    cell    field   as.col
    cell    field   as.row
    cell    field   as.CanSelect
end-struct
```

The following constants are to be used to set the various bits of the *gccp.DataType* field. The values may be combined (using `OR`) except from `GCDT-Custom` which if set causes a GCM-Rejectchar message to be sent to the Grid Control if it has the GCS_InformSelf style, otherwise it is sent to the grid controls parent.

`$0001 constant GCDT-Dec`

Accept the decimal digit characters 0,1,...9.

`$0002 constant GCDT-HEX`

Accept the decimal characters 0,1,...9 and the letters A..F. Lower case letters are accepted but include GCDT-MakeUpper to convert them to upper case.

`$0004 constant GCDT-Float`

Accepts the decimal digits 0..9 and a single decimal point (i.e. full stop) character.

`$0008 constant GCDT-Lower`

Accepts any lowercase alpha character.

`$0010 constant GCDT-Upper`

Accepts any uppercase alpha character.

`$0020 constant GCDT-SPACE`

Accepts the space character.

`$0040 constant GCDT-Custom`

Causes a GCM-Rejectchar message to be sent before accepting a character. If the Grid Control has the GCS-TalkSelf style GCM-Rejectchar is sent to the grid control otherwise it is sent to the grid control's parent window. See GCM-Rejectchar for more information.

`$4000:0000 constant GCDT-MakeUpper`
Causes lower case letters to be converted upper case.

`$2000:0000 constant GCDT-MakeLower`
Converts upper case letters to lower case.

`$1000:0000 constant GCDT-StartHL`
Causes any existing text to be highlighted when the edit box is first opened.

`$0800:0000 constant GCDT-PosStart`
Causes the caret to be positioned at the begining of any existing text when the edit box is first opened. If this flag is not returned by an application, the caret is positioned at the end.

Grid Control window messages

The following messages are sent to either a grid control's parent window or to iteslf depending on the message and in some cases the control's style.

`AppMessage: GCM-AskString`
A message sent to request for a cell's text string. wParam is the handle of the control, lParam is the address of a /GCStrspec data structure which contains the row and column numbers being considered. The application supplies the address and length of the requested string. If the control has the GCS-TalkSelf window style (defined below) this message is sent to itself, otherwise it is sent to the parent window.

`AppMessage: GCM-NewSelection`
Sent whenever a new cell is selected. That is if either the current row number and/or the current column number alters. wParam = handle of the grid control, lParamhi=column number, lParamLo=row number.

`AppMessage: GCM-QueryEdit`
Sent whenever an attempt is made to edit the current cell. This can be when the mouse is double-clicked over a cell but occurs at other times also. If the grid control has the GCS-TalkSelf style the message is sent to itself otherwise it is sent the grid control's parent window.

wParam is the handle of the control, lParam is the address of a /GCCellSpec structure (See above.) The structure's gccp.Col and gccp.Row fields are passed to the application along with the cell's bounding rectangle in gccp.CellBounds. Using this information an application can provide alternative editing facilities, such as displaying a combo box. (See *Cell Editing Overview* below).

`AppMessage: GCM-RejectChar`
This message is sent by the cell editor when the data type flag has been set to GCDT-Custom. The message is sent to the grid control's parent unless it has the GCS-TalkSelf style in which case the message is sent to the grid control.

wParam is a character code which is to be tested and wParam is the handle to the edit control (thus allowing an application to inspect the text contained). If the character is to be rejected the return value should be non-zero.

`AppMessage: GCM-CharEdit?`
A message sent by the grid control to its parent window or, if the control has the `GCS-TalkSelf` style the message is sent to itself. The message is sent whenever a character is detected (via the

WM_CHAR message).

wParam is the handle of the control, lParam is the address of a /GCCellSpec structure (See above.) The structure's gccp.Col and gccp.Row fields are passed to the application along with the cell's bounding rectangle in gccp.CellBounds. The gccp.Char field contains the character code of the key pressed. Using this information an application can provide alternative editing facilities.

This message then, works in a similar way to `GCM-QueryEdit` but in addition the text in the edit box will contain only the character represented by the key pressed.

`AppMessage: GCM-DrawCell`

The message sent when a grid control with GCS-DrawSelf style needs to draw a cell. wParam is unused. lParam is the address of a /DRAWGRIDCELL structure (see above).
NOTE: Even without the GCS-DrawSelf style, a grid control still draws itself, but the application has little control over how the drawing takes place. A sub-classed grid control can intervene in the drawing process by specifying the GCS-DrawSelf style to receive these messages.

`AppMessage: GCM-SortColumn`

Sent to either the parent window or itself when a mouse click occurs over the header control. This message is used to let the user signify that the items in the list should be ordered according to the column which was clicked. The control's parent window receives this message unless the control has the GCS-TalkSelf style in which case the control itself receives it.

The message parameters have the following meaning: wParam = hCtl, the handle of the control; lParam = col is the number of the column being clicked.

`AppMessage: GCM-AskSelect`

A message send to ask 'permission' to set the current cell. If the Grid has the `GCS-TalkSelf` style the message is sent to the grid itself, otherwisw the message is sent to the grid's parent. wParam is the handle of the grid Control. lParam is the address of an /AskSelect structure into which an application can 'answer' the question.

If an application returns non-zero in response to the message the value of the /AskSelect field as.CanSelect is taken as an answer to the request. A return value of zero implies that the message has not been answered and the column active property is used instead.

`AppMessage: GCM-TermEdit`

GCM-TermEdit is sent to inform an application that cell editing has terminated. wParam is the handle of the grid control. lParam is the address of a /GCTermEdit structure.

`AppMessage: GCM-AskTermEdit`

A message sent whenever a key is pressed while a cell is being edited (i.e. whenever a WM_KEYDOWN message is received), asking whether to terminate the editing process. wParam is the handle of the grid control. lParam is the address of a /AskTermEdit structure. An application should return non-zero to indicate that editing should be terminated. wParam is the handle of the grid control. lParam is the address of a /GCTermEdit structure.

`AppMessage: GCM-ColumnDrag`

A message sent to either the Grid Control or its parent window (depending on whether the control has the `GCS-TalkSelf` style), indicating that a column on the header control has been dragged to another position.

Grid Control styles

`: GridControlStyle:`
Defines a constant intended to be used as a control style.

`GridControlStyle: GCS-TalkSelf`
A style specifying that the grid control must send GCM-AskString, GCM-TellString and GCM-SortColumn messages to itself. If this style is not part of the Grid control those messages are sent to its parent window.

`GridControlStyle: GCS-DrawSelf`
A grid control with this style sends a GCM-DrawCell message to itself passing the address of a /DRAWGRIDCELL structure as lParam. See GCM-DrawCell above.

`GridControlStyle: GCS-ButtonHdr`
A column list style that causes the header control to have the HDS_BUTTONS style and be displayed as buttons. Only when the header control is displayed like this will GCM-SortColumn messages be sent.

`GridControlStyle: GCS-NoHdr`
Causes the control to be created without any apparent header.

`GridControlStyle: GCS-NoGrid`
Stops grid lines being drawn.

`GridControlStyle: GCS-FilterHeader`
Creates a control with a header control that has a filter bar.

`GridControlStyle: GCS-DRAGHDR`
Gives the Grid Control's header the HDS_DRAGDROP style so that column positions can now be changed.

15.2 The Cell Edit control

Cell Editing Overview.

Individual cells of a grid control can be edited by the user pressing `Enter` when the cell in question has the focus, or by double-clicking over a cell. Whenever either of these events occur, a `GCM-QueryEdit` message is sent by the grid control passing a `GCCellSpec` structure to the application.

The application has to set the `gccp.CanEdit` field of the structure to *TRUE* to allow editing to occur. If the application does not respond to the message, no further action is taken (i.e. editing does not occur).

Typically an application inspects `gccp.Col` and `gccp.Row` in deciding whether a cell may be edited and sets gccp.CanEdit accordingly. The other fields allow the application to control some aspects of the editing process: `gccp.MaxChars` is set by the application to limit the maximum number of characters allowable in that cell. `gccp.DataType` is pre-set to zero and indicates that any typable character is acceptable. The non-zero bits of `gccp.DataType` control which characters will be allowed. For instance the bit value $0001 indicates that decimal digits are allowed. See the *GCDT-* constants described above.

The GCM-QueryEdit message is sent by default to the grid control's parent - most commonly a dialog box. If however the grid control has the `GCS-TalkSelf` style the message is sent to itself.

Editing a cell involves placing an edit control over the cell and giving it the keyboard focus. Editing is completed by pressing `Enter`, clicking on another part of the grid control or pressing `Cancel`, in which case the edit control is closed and focus returns the the grid control. If editing was terminated by `Cancel` no further action is taken. If editing was ended in any other way a `GCM-TellString` message is sent to the application and the cell is redrawn. (`GCM-TellString` is sent to the parent of the grid control, or in the case that it has the `GCM-TellString` Style it is sent to itself).

Another editing possibility remains: an application can respond the the GCM-QueryEdit message ignoring the GCCellSpec structure and providing a completely different interface. This idea has not been explored but for an application that wished to place a different control over the cell, the coordinates of its bounding rectangle in the `gccp.CellBounds` is provided.

Cell Editing words

The following words are all internal to the grid control and will not be used by applications.

`: IsAllowed?  \ char u -- flag`
Returns TRUE iff the character *char* is allowed according to the GCDT-xxx bit-flags in u.

`AppMessage: GCM-SetProp`
A message used internally to set various control properties, such as setting the number of columns a grid control has. An application need not use this since a complete set words is provided to set the properties.

`AppMessage: GCM-GetProp`
A message used internally to get various properties of a grid control such as the number of columns a grid control has. Intruduced in 2009, all 'getter' words are now directed through this message.

`AppMessage: GCM-KeyMove`
Internal message sent to the edit control's parent (i.e. the grid control) when a cursor movement key has been pressed and the edit box should move accordingly. The keys detected are Up, Down, Page Up and Page Down. wParam is the virtual key code (a Windows VK_xxx constant), lParam is unused.

`AppMessage: GCM-EditCell`
An internal message which causes the 'edit cell' process to begin.

`AppMessage: GCM-SCROLLED`
Sent whenever a Grid Control is scrolled. lParam is unused. wParamLo is the ID of the control and wParamHi indicates the direction of scroll: 0 is horizontal, 1 is vertical. The Grid Edit control extended winprops are defined as follows:

```
WinProps: /GridEdit
\    cell field      gce.Cancelled   \ TRUE if editing is cancelled
    cell field      gce.EditBits    \ bits controlling allowed chars
\    cell field      gce.TermKey     \ char which ended editing, or zero
    cell field      gce.LastKey     \ Last key pressed
    cell field      gce.CtrlDown    \ TRUE if the Control key is pressed
    cell field      gce.ShiftDown   \ TRUE if the the Shift key is pressed
end-struct


: KeyMoveEdit   \ keycode --
```

Sends a `GCM-KeyMove` message to the parent window. A factor of the WM_KEYDOWN message handler.

`: EProc-KeyDown  \ --`
Process a key down event for the edit control within the grid This word is called in response to a WM_KeyDown notify event.

`defer EProc-KillFocus  \ --`
The action taken when the control looses focus and editing is deemed to have finished. See later for (EProc-KillFocus)

`: Eproc-SetFocus  \ --`
The action taken when the control gains focus. Performs some initialisation of data structures.

`: Eproc-GetDlgCode  \ --`
The WM_GETDLGCODE handler. Returns a value to allow the detection of the cursor control keys - the left and right arrow keys for example.

`: EProc-Custom?  \ -- flag`
If the edit controls WinProps field *gce.EditBits* does not have the `GCDT-Custom` bit set FALSE is returned.

If the `GCDT-Custom` bit is set a `GCM-RejectChar` message is sent to either the parent grid control if it has the *\i{GCS-TalkSelf) style or to the grid contol's parent otherwise.

A `GCM-RejectChar` handler should return TRUE for each character to be rejected and FALSE for each allowed character. If no handler is supplied all characters are allowed since the value returned by **SendMessage** in such a case is zero.

*\bo{Note:} This style may be combined with any other of the *GCDT-xxx* constants in a meaningful way. For instance to allow digits, lower-case character and commas, combine the `GCDT-Lower`, `GCDT-Dec` and `GCDT-Custom` bits and supply a `GCM-RejectChar` handler which accepts only commas.

`: EProc-AcceptChar?  \ -- flag`
Returns TRUE if the character in wParam is acceptable according to the bits set in the control's *gce.EditBits* field (which is the value supplied in the *\i{gccp.DataType) field supplied by a `GCM-QueryEdit` message.

`: EProc-AcceptChar?  \ -- flag`
Returns TRUE if the character in wParam is acceptable according to the bits set in the control's *gce.EditBits* field (which is the value supplied in the *\i{gccp.DataType) field supplied by a `GCM-QueryEdit` message.

`: Eproc-Char  \ --`
The WM_CHAR handler. Its action depends on the value of the control's gce.EditBits value. If it is zero the Windows default processing is performed. If gce.EditBits contains the bit GCDT-Custom a GCM-Rejectchar message is issued. The other bits specify various character checking filters and if the character is accepted the default Windows process is performed.
The edit control used is a subclassed Windows-supplied edit control. It is single line control, and usually in not visible.

The GridEdit control is defined below:

```
WinControl: GridEdit "edit"
    /GridEdit PropertySize
```

```
    WS_CHILD                Style
    WS_VISIBLE              Style-
    WS_TABSTOP WS_BORDER or Style+
    ES_LEFT                 Style+
    ES_AUTOHSCROLL          Style+

    WM_KeyDown      does Eproc-KeyDown
\    WM_KEYUP        does EProc-KeyUp
    WM_KillFocus    does Eproc-KillFocus
    WM_SETFOCUS     does Eproc-SetFocus
    WM_GETDLGCODE   does Eproc-GetDlgCode
    WM_Char         does Eproc-Char

    clr-Window Colour
end-Control
```

15.3 The Grid Control

Grid Control properties

```
100 Constant gc-MaxCols
```

The maximum number of columns a Grid Control may have.
The extended *WinProps* of the grid control contains a number of fields. Easy access to the the header control handle is provided by `gc.hHdr`. The other fields hold values unobtainable elsewhere.

GridProps is defined below:

```
struct  /GridProps    \ data structure stored with each instance
    /WinProps +
    Cell Field gc.hHdr          \ handle of grid's header control
\    Cell Field gc.hEd           \ handle of grid's Edit control
    Cell Field gc.#rows         \ number of rows
    Cell Field gc.CurCol        \ current column number
    Cell Field gc.CurRow        \ current row number
    cell field gc.BgdBrush      \ brush used to fill background

    Cell Field gc.LineHeight    \ height of a line/row
    Cell Field gc.HdrHeight     \ printing starts below this
    cell field gc.Top           \ top line shown below the header
    cell field gc.VSCroll       \ flag indicating if scroll showing
    cell field gc.HSCroll       \ flag indicating if scroll showing
    RECT field gc.ScrollRect    \ rectangle used to scroll
    gc-MaxCols 2 cells
            array-of gc.refs    \ application-defined 64-bit values
    gc-MaxCols cell
            array-of gc.Active  \ column 'active' flags
end-struct
```

Some "primitives"

The following words provide some low level functions to access the grid control properties and related values.

`: #Cols \ -- n`
Returns the number of columns in the grid control.

`: #Cols! \ n --`
Sets the number of columns in the grid control.

`: #rows \ -- n`
Returns the number of rows in the grid control.

`: #rows! \ n --`
Sets the number of rows in the grid control.

`: Col#! \ n --`
Sets the current column number.

`: Col# \ -- n`
Returns the current column number.

`: Row#! \ n --`
Sets the current row number.

`: Row# \ -- n`
Gets the current row number.

`: Top \ -- n`
Gets the current top row number.

`: Top! \ n --`
Sets the current top row number.

`: clear \ --`
"Empties" the grid control. Sets the both the number of columns and the number of rows to zero, and both the current row and current column to -1.

`: Ref# \ col# -- u`
Returns the 32-bit reference number for the column number given.

`: XOffset \ -- n`
Returns the current horizontal offset of the grid. The horizontal offset is the amount by which the grid columns are drawn 'displaced'. When the grid control is too wide for all columns to be shown, the view can be *scrolled left* in which case this value becomes negative.

`: XOffset! \ n --`
Sets the value of the horizontal offset.

`: ColWidth \ col# -- width`
Returns the width of column number *col#*.

`: ColPos \ col# -- n`
Find the horizontal position of the given column number relative to the leftmost position. No account is taken of the current horizontal offset - this is NOT the pixel position within the control. In other words *n* is the sum of the first *col#* column widths.

`: ColLeft \ col# -- left`
The horizontal position in client coordinates of the leftmost side of the given column.

`: LineTop \ line# -- top`
The vertical position of the top of line number *line#*.

`: ColWidth! \ n col# --`
Sets the width of the given column number.

`: x>col \ x -- col#`
Convert a screen client x-coordinate to a column number.

`: y>row \ y -- row#`
Convert the given client y-coordinate to a row number.

`: EqualWidths \ --`
Make all columns of equal width. Performed only in response to the corresponding *Set Property* message. (See the `GCM-SetProp` message handler.

`: VSCroll! \ flag --`
Set the vertical scroll flag to the given value.

`: VSCroll? \ -- flag`
Return the value of the vertical scroll flag.

`: HSCroll! \ flag --`
Set the horizontal scroll flag to given value.

`: HSCroll? \ --`
Get the horizontal scroll flag.

`: ValidCell? \ col# row# -- flag`
Returns TRUE iff the given column and row numbers specify a valid cell. That is *col#* must greater than or equal to zero but less than the number of columns and *row#* must be greater than or equal to zero but less than the number of rows.

Messaging 'primitives'

`: InformWhom? \ -- hWnd`
Returns the handle of the window which is to receive GCM-NewSelection and GCM-QueryEdit messages. If the current window has the `GCS-TalkSelf` style the result is the current window handle otherwise the handle of the current window's parent window is returned.

`: CellChanged \ --`
Send a GCM-NewSelection message to the current control's parent window or if the control has the GCS-TalkSelf style, the message is sent to itself. wParam is the control's handle, lParamHi is the new current column number, lParamLo is the new current row number.

Creating the Grid Control

`: GClass-ChkHdrBtns \ --`
Called during creation, the grid control's style bit flags are inspected and if GCS-ButtonHdr is one of them HDS_BUTTONS is added to the header control's style causing the header sections to be displayed as buttons.

`: GClass-DefLineHt \ -- n`
Calculate a default line height from TextMetric data. Used by `GClass-Create` to set the win-props field gc.LineHeight.

`NextID: IDC_Header`
The ID of the grid controls header control.

`NextID: IDC_Edit`

The ID of the grid controls edit control.

`: GClass-Create \ --`
The WM_CREATE window message handler. Creates the header control and the edit control and initialises the GridProps fields.

Drawing the gridlines

`: LinesSeen \ -- n`
Get number of lines on screen (including any fractional lines).

`: /page \ -- n ;`
Get number of lines on screen excluding any fractional lines.

`: HLine \ xmax ypos --`
Draw a line from (0,ypos) to (xmax,ypos) on the current DC.

`: VLine \ ymax xpos --`
Draw a line from (xpos,0) to (xpos,ymax) on the current DC.

`: HLines \ --`
Paint the horizontal grid lines.

`: ?Vline \ ymax x --`
Draw the vertical line from (x,0) to (x,ymax) iff x is greater then zero. Used by `VLines`. The point of this is to avoid drawing lines which will not be shown or seen.

`: VLines \ --`
Draw the vertical grid lines.

Painting the Grid control

`: DrawSelf? \ -- flag`
Returns TRUE iff the (current) window style includes `GCS-DrawSelf`.

`: ColLeftRight \ col# -- left right`
Returns x-coordinates of the left hand and right hand sides of column number *col#*.

`: CellBounds \ col# line# -- left top right bottom`
Returns the coordinates in pixels, of the bounding rectangle to the cell at the given column and line numbers.

`: CellDims \ col# line# -- left top width height`
Returns the "dimensions" in pixels of the specified cell.

`: ExpandDims \ x y w h n -- x' y' w' h'`
Expand the rectangular area defined by the given position (x and y) and size (w and h). x and y are reduced by n, while w and h are increased by 2n.

`: CellInner \ col# line# -- left top right bottom`
Returns the rectangle coordinates of the specified cell's inner or clipping rectangle. Text is to be drawn within this rectangle. (Units in pixels).

`: GClass-GetStrSpec \ col# row# strspec --`
Retreive the *string specification* for the specified cell. Sends a `GCM-AskString` message to either the parent or itself as decided by `AskWho?`.

This word is exported as GCproc-GetStrSpec to help in creating application-specific editing functions.

`: GClass-Fill-DGC  \ line# col# row# DRAWGRIDCELL --`
Fill the supplied DrawGridCell structure with information sufficient to draw a cell.

`: dgc-ColourCell  \ dgc --`
Given the address of a DRAWGRIDCELL structure, the bounding cell rectangle specified by the dgc.CellRect field is filled with the background colour specified by the dgc.GCss fields GC-Strspec.BkgdColour subfield.
NOTE: this field is `EXPORT`ed for used by applications answering the GCM-DRAWCELL message.

`: dgc-DrawText  \ dgc --`
Draws a cell's text as specified by the given DRAWGRIDCELL structure. The current text colour is not altered. (Compare `dgc-ColourText`).
NOTE: this field is `EXPORT`ed for used by applications answering the GCM-DRAWCELL message.

`: dgc-ColourText  \ dgc --`
Draws a cell's text in the colour specified in the DRAWGRIDCELL structure given.
NOTE: this field is `EXPORT`ed for used by applications answering the GCM-DRAWCELL message.

`: dgc-PaintCell  \ dgc --`
The paint cell word. Decides how the text in the DRAWGRIDCELL structure given is drawn : if coloured text is specified in DRAWGRIDCELL `dgc-ColourCell` and `dgc-ColourText` are executed, otherwise `dgc-DrawText` is executed.
NOTE: this field is `EXPORT`ed for use by applications answering the GCM-DRAWCELL message.

`: GClass-PaintCell  \ line# col# row# --`
The 'standard' *paint cell* method : the cell's text is drawn without sending a `GCM-DrawCell` message.

`: GClass-PaintLine  \ n --`
Paint line number n. The line number is the line displayed and will show row number n+gc.Top. This is the 'standard' *paint line* method, executing `GClass-PaintCell` repeatedly for each cell on line n.

`: GClass-PaintCellSelf  \ line# col# row# --`
The 'none-standard' method of painting the cell : a `GCM-DrawCell` message is sent to itself. If the result of sending the message is zero `dgc-PaintCell` is called.

`: GClass-PaintLineSelf  \ line# --`
Paint line number line# by sending a GCM-DrawSelf message for each visible cell. This is the 'non-standard' *paint line* method which repeatedly exexcutes `GClass-PaintCellSelf`.

`: OutlineCell  \ col# row# --`
Draws a line arround the specified cell. This is the word used to show the 'current cell'.

`: LinesToPrint  \ -- n`
Return the number of lines to draw.

`: PaintSelf  \ --`
The main paint routine for a grid control with the `GCS-DrawSelf` style. `EXECUTE`s `GClass-PaintLineSelf` for each line displayed.

`: PaintHere  \ --`
The main paint routine for a grid control without the `GCS-DrawSelf` style. Repeatedly `EXECUTE`s `GClass-PaintLine` for each line shown.

`: GreyPen  \ -- hPen`
Creates a sold 'pen' which is used to draw the 'grid lines'.

```
: GClass-Paint  \ --
```

The WM_PAINT handler. Calls `PaintHere` or `PaintSelf` accoring to the result of `DrawSelf?`.

```
: Draw-Cell     \ col# row# --
```

Erase and draw the current cell. Not part of the WM_PAINT procedure - a new DC is opened. The word is used by `Set-Cell`.

```
: Hilite-CurrCell   \ --
```

Outline the current cell if this window has focus. This is not done as part of the WM_PAINT procedure - a new DC is opened. The word is used by `Set-Cell`.

Managing the scroll bars

```
: RqdWidth  \ -- n
```

Return the client widph required to show all columns.

```
: RqdHeight \ -- n
```

Return the client height required to show all rows.

```
: VSCroll-Off   \ --
```

Turns the ver|ical scroll bar off (i.e. hides it) if not already shown and sets the top line to be line 0.

```
: VScroll-On    \ --
```

Turns the vertical scroll bar on (i.e. shows it), if it is not already shown, and initialises the verical scroll bar.

```
: MustVSCroll?  \ -- flag
```

Returns true if a vertical scroll bar is needed regardless of there being a horizontal scroll bar shown or not.

```
: NotVSCroll?   \ -- flag
```

Test if a vertical scroll bar is NOT needed, that is if all rows will fit even if there is a horizontal scroll shown.

```
: MustHSCroll?  \ -- flag
```

Return *TRUE* iff a horizontal scroll bar **must** be shown - that is if not all columns will fit on the control without a vertical scroll bar.

```
: HScroll-On    \ --
```

Show the horizontal scroll bar if required, and initialise the scroll bar.

```
: ?FitHeader    \ --
```

increase or decrease the size of the header to fit the client width.

```
: HScroll-Off   \ --
```

Hide the horizontal scroll bar (if not already hidden), and set the horizontal scroll value (i.e. the *XOffset* value to zero).

```
: ?HScroll
```

Check the horizontal bar: hide it if it is not required. If it is required then show it (if it is not already shown), and adjust the range and button size.

```
: ?Scrolls  \ --
```

Check the scroll bars. That is show them if required and hide them if not.

Handling header item events

The header control comunicates with the column list via the Windows WM_NOTFY message. The following words prefixed by the characters *HdrCtl-* service that message.

`: HdrCtl-AdjustWidth \ --`
Adjust the width of the header control to match the width if its items. No action if no adjustment necessary.

`: HdrCtl-ItemChanged \ --`
Handler for the header control WM_NOTIFY/HDN_ITEMCHANGED event. `HdrCtl-AdjustWidth` is called to set the header control width, scroll bars are added to or removed from the grid control as necessary and the control is repainted.

`: HdrCtl-HdrClick \ --`
The action performed in response to a HDN_ITEMCLICK notification message from the header control. The header must have the HDS_BUTTONS style to generate this message. The single action performed is to send a GCM-SORTCOLUMN message the parent window if HDN_ITEMCLICK was caused by a left button click. If a non-zero result is returned the window is re-painted.
wParam is the handle of the grid control, lParam is the column number.

`: HdrCtl-WM_NOTIFY \ --`
Handle the current WM_NOTIFY message that came from the header control

`: GClass-WM_Notify \ --`
The grid control WM_NOTIFY message handler. The only notification messages come from the header control.

Handling Scroll bar events

`: ALot \ -- u`
The size in pixels of the horizontal scroll change which occurs when large or 'page' scroll occurs.

`: ABit \ -- u`
The small horizontal scroll change amount in pixels.

`: Set-XOffset \ n -- ;`
Set the current XOffset. The number n is negative or zero. Causes the window to be repainted if the resulting XOffset has changed.

`: ScrollLeft \ u --`
Scroll left by u pixels by adding u to the current XOffset.

`: ScrollRight \ u --`
Scroll left by u pixels by subtraction u from the current XOffset.

`: Horz-ThumPos \ --`
The SB_THUMBTRACK scroll message action which is to set the horizontal scroll to the (negative) of the value of wParamHi.

`: GClass-HorzScroll \ --`
The gridcontrol WM_HSCROLL message handler.

`: ScrollLines \ n --`
Performs a 'smooth scroll' of n lines

`: set-VSCroll \ n --`
Sets the current vertical scroll value to 'n lines'.

```
: ScrollDown  \ n --
```
Scroll down by n lines.

```
: ScrollUp    \ n --
```
Scroll up by n lines.

```
: GClass-VertScroll     \ --
```
Handles the vertical scroll bar message.

Setting the grid control's properties

The properties are set using a GCM-SetProp message, with lParamLo = property number. lParamHi and wParam are defined according to the property number. The following property numbers are defined :

- lParamLo = 0 (Set number of columns), wParam is the number of columns.
- lParamLo = 1 (Set number of rows), wParam is the number of rows.
- lParamLo = 2 (Set column width), wParamLo is column number, wParamHi is width.
- lParamLo = 3 (Clear/Empty the list), wParam is unspecified.
- lParamLo = 4 (Set equal column widths), wParam is unspecified.
- lParamLo = 5 (Set current row), wParam is row number.
- lParamLo = 6 (Set current col), wParam is col number.
- lParamLo = 7 (Set current cell), wParamHi is col, wParamLo is row.
- lParamLo = 8 (Set top row), wParamHi is row.
- lParamLo = 9 (Set line height), wParamHi is line height.
- lParamLo = 10 (Set title), wParam=zString, lParamHi=col#
- lParamLo = 11 widen/narrow columns to fit. wParam, lParamHi unused
- lParamLo = 12 (Draw Cell) wParamLo=col, wParamHi = Row. lParamHi unused
- lParamLo = 13 (Set column active), wParamLo is column number, wParamHi is flag.
- lParamLo = 14 (move column), lParamHi is direction code
- lParamLo = 15 (Set column widths) wParam = address of array of widths

```
: CurrCell?     \ col# row# -- flag
```
Given a cell and row numbers, return TRUE iff they are equal to the current cell and current row numbers.

```
: SensibleCell  \ col# row# -- col#' row#'
```
Return the column and row numbers adjusted to be within the ranges [0 .. #cols[and [0 .. #rows[respectively.

```
: PlaceLeft     \ col# --
```
Position the given colum on the left of the client area by changing the horizontal scroll value.

```
: PlaceRight    \ col# --
```
Change horizontal scroll so that the given column is at right of screen.

```
: BringInCol    \ --
```
Adjust the horizontal scroll to show column number col# within the grid control.

```
: BringInRow    \ --
```
Move current row into view if it not already there. This is accomplised by changing the vertical scroll position.

```
: OutWindow?    \ -- flag
```

Is current column outside of the window?

`: Set-Cell \ col# row# --`
Set the current column number and row number. If a change in either value results a GWM_NewSelection message is sent.

`: Set-Column \ n --`
n is first limited to fall between 0 and #cols-1 and the current column number set to the result. If the new current column number is different a GWM_NewSelection message is sent.

`: Set-Row \ n --`
Set the current row number to n and sends a GWM_NewSelection message. If the current rot number is already equal to n no action is taken.

`: GClass-SetProp \ --`
The GCM-SetProp message handler
Grid Control properties are obtained using the GCM-GetProp message, with lParamLo = property number. lParamHi and wParam are defined according to the property number. The following property numbers are defined :

- 0 : Get number of columns
- 1 : Get number of rows
- 2 : Get column width. wParamHi is column number
- 3 : unused
- 4 : unused
- 5 : Get current row number.
- 6 : Get current colunm number.
- 7 : Get current row and column numbers; row in highest 16 bits
- 8 : Get top row number.
- 9 : Get line height.
- 10 : Get column title. lParamHi=col#
- 11 : unused
- 12 : unused
- 13 : Get column active flag. wParamLo is column number.
- 14 : unused
- 15 : Get column reference field address
- 16 : Get the column given by x client coord in wParam
- 17 : Get the row given by y client coord in wParam

The WM_SIZE message handler

`: GClass-WM_Size \ --`
The word executed in response to a WM_SIZE message. Checks the horizontal and vertical scroll states, adding or removing them as requiresd and the sets the size of the control's scrolling rectangle `gc.ScrollRect`.

Editing a cell

`: GClass-AskEdit \ GCCellSpec -- flag`
Send the GCM-QueryEdit to ask if the current cell can be edited. flag is TRUE if editing is allowed in which case the GCCellSpec structure will have been filled in with the relavent details (i.e. type of edit and maximum number of characters).

`: GClass-InFromRight \ left top width height -- left' top width height`
Adjust the input coordinates so that the rectangle they represent does not cross the right hand side of the control.

`: GClass-InFromLeft \ left top width height -- left' top width height`
Adjust the input coordinates so that the rectangle they represent does not cross the left hand side of the control.

`: GClass-EditDims  \ -- left top width height`
Return the 'dimensions' with which to display the edit box.

`: GClass-SetEditStr \ --`
Get the cell string and set the edit box text to that value.

`: GClass-PrepEdit  \ --`
Show the edit control positioned over the cell being edited and with the correct width and height.

`: GClass-SetupEdit  \ GCCellSpec --`
Set the edit control properties according to the data held in the /GCCellSpec structure.

`: GClass-Edit  \ --`
Attempt to edit the current cell. Sends a GCM-QUERYEDIT message and if a 'no' answer is returned, no action is taken otherwise the editor is 'opened'.

`: GClass-Edit  \ --`
Attempt to edit the current cell. Sends a GCM-QUERYEDIT message and if a 'no' answer is returned, no action is taken otherwise the editor is 'opened'.

`: (EProc-KillFocus) \ --`
The action taken when the control looses focus. The edit window is closed and a GCM-TermEdit message is sent to the parent grid control or the grid control's parent (depending on the grid's GCS-TalkSelf style). See the information for GCM-TermEdit and the /gcTermEdit structure.

Grid Control Key handling ...

The grid control accepts some keyboard input: cursor keys to move the current 'focus' to another cell and `Enter` to edit the cell. Since the grid control is a application defined control intended to be used 'inside' a dialog box, the WM_GETDLGCODE message is used to tell the the dialog box which keys presses it wants knowledge of.

`: GClass-GETDLGCODE  \ --`
WM_GETDLGCODE message handler. Informs the parent dialog that the control wants to detect the cursor control keys and the tab key in addition to any others.

`: RowsDown  \ n --`
Moves the current row down by n rows. Uses `Set-Row`.

`: RowsUp  \ n --`
Moves the current row up by n rows. uses `RowsDown`.

`: GClass-KeyDown  \ --`
The WM_KeyDown message handler. Uses the preceding words to move the focus to another cell.

`: GClass-KeyMoveEdit  \ --`
Response to the GCM-KeyMove message which is sent by the edit control when a cursor control key has been pressed and the current cell should change. wParam is the key code of the key pressed. lParam is unused.

Handling mouse button messages

`: x>col \ x -- col#`
Convert a screen client x-coordinate to a column number.

`: y>row \ y -- row#`
Convert the given client y-coordinate to a row number.

`: GClass-LBUTTONDOWN \ --`
The WM_LBUTTONDOWN message handler. Sets the current cell to be that cell under the pointer.

`: GClass-LDblClick \ --`
WM_LBUTTONDBLCLK message handler. Enables the editor for the current cell by calling `GClass-Edit`. If the mouse does not lie over a cell no action is taken.

`: GClass-SetFont \ -- ( 002:Feb-06)`
The WM_SETFONT message handler. Posts that same message with the same parameters to the header control.

`: GClass-EditCell \ -- ;`
The response to the GCM-EditCell message. Calls the GCM-QueryEdit message to begin editing a cell as though the user had pressed Enter.

`: GClass-AskCharEdit \ GCCellSpec -- flag`
Send the GCM-CharEdit? to ask if the current cell can be edited. flag is TRUE if editing is allowed in which case the GCCellSpec structure will have been filled in with the relavent details (i.e. type of edit and maximum number of characters). The field gccp.Char contains the character code of the key pressed and an application can check this to help determine if the cell indicated (in gccp.Col and gccp.Row) can be edited.

`: GClass-Char \ -- ;`
The WM_CHAR message handler. If the character in wParam (the key pressed) is 'allowed' for that column, the message GCM-CharEdit? is sent. If the gccp.CanEdit field of the /GCCellSpec structure sent with that message is true, the current cell will be edited in the normal way.

`: GClass-Wheel \ -- ; WM_MOUSEWHEEL message handler`
The WM_MOUSEWHEEL message. Simply does a 1-line scroll for each wheel rotation unit.

The GridClass Window class and the GridControl control.

The grid control is an 'application control' - it is created by the application rather than being provided by Windows. Like all application conttrols, the grid control is based on a window class, in this case the class is `GridClass`. The GridClass with its window procedure provide the main behaviour - painting, cell editing, cursor movement etc.

For instructions on how to use a grid control see the section *Using the Grid Control* later.

The GridClass and the GridControl are defined as follows:

```
WinClass: GridClass
    /GridProps    PropertySize
    Name "GridClass-GCS"
    CS_HREDRAW CS_VREDRAW or    Style
    CS_DBLCLKS                  Style+

    WM_CREATE           does GClass-Create
    WM_PAINT            does GClass-Paint
```

```
    WM_ERASEBKGND       does GClass-Erase
    WM_ENABLE           does GClass-Enable
    WM_SIZE             does GClass-WM_Size
    WM_NOTIFY           does GClass-WM_Notify
    WM_LBUTTONDBLCLK    does GClass-LDblClick
    WM_VSCROLL          does GClass-VertScroll
    WM_HSCROLL          does GClass-HorzScroll
    WM_GETDLGCODE       does GClass-GETDLGCODE
    WM_KeyDown          does GClass-KeyDown
    WM_LBUTTONDOWN      does GClass-LBUTTONDOWN
    WM_SETFONT          does GClass-SetFont
    WM_SETFOCUS         does Hilite-CurrCell
    WM_KillFocus        does LoLite-CurrCell
    WM_CHAR             does GClass-Char
    WM_MOUSEWHEEL       does GClass-Wheel   \ 17Feb11

    \ now processs my own codes...
    GCM-KeyMove         does GClass-KeyMoveEdit
\ 20Feb09    GCM-EndEdit         does GClass-EndEdit
    GCM-SetProp         does GClass-SetProp
    GCM-GetProp         does GClass-GetProp
    GCM-EditCell        does GClass-EditCell

End-WinClass


: ?gc   gridclass .msgchains ;



GridClass  AppControl: GridControl
    WS_CHILD WS_BORDER or       Style
    WS_HSCROLL WS_VSCROLL or    Style+
    WS_TABSTOP                  Style+

    clr-Window Colour
End-Control
```

15.4 Application words

The following words provide the interface between the grid control and applications using it. All these words are prfixed by "GC-" to indicate *G*rid *C*ontrol.

Setting properties ...

`: gc-SetCols    \ n hGC --`
Set the number of columns to n

`: gc-SetRows    \ n hGC --`
Set the number of rows to n

`: gc-SetColWidth    \ n hGC col# --`
Set the column width to n

`: gc-SetColActive \ flag hGC col# --`
Set the column width to 'active' flag

`: gc-clear \ hGC --`
Clear the grid control. After this operation the number of rows and columns on the grid is zero

`: gc-EqualWidths \ hGC --`
Sets the widths of each column to the same value such that all columns fit inside the client area of the control.

`: gc-FitWidths \ hGC --`
Adjusts the widths of each column proportionatly so that the all columns fit inside the client area of the control.

`: gc-SetRow# \ n hGC --`
Set the current Row number to n

`: gc-SetCol# \ n hGC --`
Set the current column number to n

`: gc-SetCell \ col# row# hGC --`
Set the current cell

`: gc-SetTop \ n hGC --`
Set the top row number to n

`: gc-SetLineHeight \ n hGC --`
Set the line height to n

`: gc-SetTitlez \ zaddr hGC col# --`
Set the title of the given column number. This is done by sending a GCM-SetProp message.

`: gc-SetTitle \ caddr ulen hGC col# --`
Set the title of the given column number. Uses `gc-SetTitlez`.

Getting properties ...

`: gc-AddCol \ caddrz width uref hGC --`
Add a new colum to the grid control with handle *hGC* of *chunits* horizontal character units wide (See `HUnits`), and setting the column title to the string ad address *\{caddrz} which is a zero-terminated string. *uref* is a 32-bit value which is stored at the new columns reference address such that the value can be recovered using `gc-ref @`.

`: gc-AppendCol \ ca u width ref hGC --`
Add a new colum to the grid control with handle *hGC* of *width* pixels wide and setting the column title to the string specified by the ca/u pair. *uref* is a 32-bit value which is stored at the new columns reference address such that the value can be recovered using `gc-ref @`.

`: gc-EditCell \ hGC --`
Causes the process of editing the current cell starting with the sending of a `GCM-QueryEdit` message to the application. Note that this word will return (probably) before the message is sent.

`: gc-HdrHandle \ hGC -- hHdr`
Returns the handle of the gridcontrol's header control

`: gc-HdrID \ -- u`
Returns the ID of the gridcontrol's header control

`: gc-InitColGrid \ table hGC --`

Initialise a grid control given the address of a Gridcontrol Initialsation Table and the grid control's handle.

The Gridcontrol Initialsation Table follows the following structure:

```
   Create MDlgGrid
```

`: gc-FitToTable \ table hGC --`
Adjusts the column widths to be spaced accross the width of the control in proportion to the widths specified in the grid initialisation table.

Help for applications using a Grid Control

`GClass-EditDims  \ -- left top width height`
For use by applications within a Grid Control message handling procedure. Returns the position and dimensions which are used when positioning an edit control over the current cell. The position may not lie directly over the cell, it is adjusted in an attempt to ensure the rectangle described does not overlap the grid control edge.

`GClass-TellString  \ addr len --`
This is the word which sends a GCM-TellString message to the appropriate window. Applications can use it at the end of a custom cell editing process within a grid control message handler routine only.

15.5 Using the Grid Control

The following example code shows the use of a grid control, illustrating techniques for

- 1) Painting cells in different ways.
- 2) Editing some cells but not others.

The example creates a sub-classed grid control which has the `GCS-TalkSelf` and `GCS-DrawSelf` styles. This allows the control to handle its own cell data handling including painting cell text.

The grid will show the contents of a 2-dimensional array which is a simple dictionary definition defined here along with a word to access individual elements of it.

```
4   constant #Columns
50  constant #Rows
31  constant MaxTxtLen
Create Table
    #Columns #Rows * 31 1+ * Allot&Erase

: table[]   \ col# row# -- addr
    #Columns * + 31 1+ * table +
;
s" 0" 0 0 table[] place
s" 2" 0 2 table[] place
s" 4" 0 4 table[] place
s" 6" 0 6 table[] place
s" 8" 0 8 table[] place
s" 10" 0 10 table[] place
s" 12" 0 12 table[] place
```

Words to supply a grid control with the table's element text, to change the table's element text and to draw a cell are defined next.

The following words provide the message handling for the new sub-classed gridd control.

```
: SupplyText    \ --
\ The response to a GCM-AskString message. lParam is the
\ address of a GCStrspec structure which is filled in
\ with the address of the text for the specified element.
    \ locate the element concerned...
    lParam GCStrspec.Col @ lParam GCStrspec.row @ table[]
    \ and tell the grid control what its text is...
    count lParam GCStrspec.len ! lParam GCStrspec.text !
;

: DrawCell      \ --
\ The gcm-DrawCell message handler. Draws the cell
\ specified in the DRAWGRIDCELL structure passed in lParam.
    lParam dgc.GCss             \ addr ; get the GCStrspec structure
    dup GCStrspec.col @ 2 <> if drop exit then

    RED over GCStrspec.TextColour !
    Yellow over GCStrspec.BkgdColour !
    GCStrspec.Coloured on
    lParam dgc-PaintCell
;

: InitMyGrid    \ --
\ The GWM_Initcontrol message response.
    #Columns currwin gc-Setcols
    #Rows currwin gc-Setrows
    CurrWin gc-EqualWidths
;

: SortColumn    \ --
\ The GCM-SortColumn message handler - the word performed
\ when the user clicks on the header.

    \  Fill this word in to sort the data ...
;

0 value WorzgVal

: QueryEdit     \ --
\ The GCM-QueryEdit message handler. Allows any cell in columns
\ 0, 1 and 2 to be edited. It the column is number two only 5
\ digit characters are allowed, otherwise up to 31 characters
\ of any type are allowed.
```

```
    \ Get out if the column is greater than 2
    lParam gccp.Col @ 2 > if exit then

    True lParam gccp.CanEdit !      \ otherwise editing is allowed.

    case lParam gccp.Col @
    1   of  WorzgVal lParam gccp.DataType !
            20 lParam gccp.MaxChars !
        endof
    2   of  \ GCDT-Lower GCDT-Dec or ( GCDT-Custom or) lParam gccp.DataType !
            GCDT-Dec GCDT-Custom or lParam gccp.DataType !
            5 lParam gccp.MaxChars !
        endof
    endcase

;

: ?CharEdit     \ -- ; GCM-CharEdit? message handler
\ allow editing if a number is typed in column two
    lParam gccp.Col @ 2 <> if exit then
    lParam gccp.Char @ [char] 0 [char] 9 within?
    if QueryEdit
        lParam gccp.Char off
    then
;

: ?TermEdit     \ -- ; GCM-QueryTerm handler
\ lParam is a /GCTermEdit structure.
    inherited       \ do the default ...
    lParam termedit.TermKey @ VK_Return =   \ flag ; want a return key
    ReturnVal or!   \ add it to the ReturnVal value
;

: TermEdit      \ -- ; GCM-TermEdit handler
\ The GCM-TermEdit message handler. The text
\ entered by the user when a cell grid is edited is
\ returned to the application here where it is stored
\ in the table.
\ lParam is /GCTermEdit struct

    \ We will accept text ONLY if editing was terminated
    \ by the <Enter> key..
    lParam termedit.TermKey @ 13 <> if exit then

    \ OK, now we get the text ...
    lParam termedit.Text zcount MaxTxtLen min

    \ and now store in the table ...
    lParam GCStrspec.Col @ lParam GCStrspec.row @ table[]
    place
;
```

```
Another GridControl MyGrid
    GCS-TalkSelf GCS-DrawSelf or GCS-TalkSelf or GCS-ButtonHdr or Style+

    GCM-AskString   does SupplyText
\    GCM-TellString  does ReceiveText
    GWM_Initcontrol does InitMyGrid
    GCM-SortColumn  does SortColumn
    GCM-QueryEdit   does QueryEdit
    GCM-DrawCell    does DrawCell
    GCM-RejectChar  does: wParam [char] , <> ReturnVal ! -inherited ;
    GCM-CharEdit?   does ?CharEdit

    GCM-QueryTerm       does    ?TermEdit
    GCM-TermEdit        does    TermEdit

end-control
```

The dialog box is now defined. An auxilliary word is defined to correctly size the grid control within the dialog box.

```
NextID: IDC_Grid
\ The grid control's numeric ID required by Windows.

: textbox-posgrid   \
\ The word to position the grid control within the dialog box.
    CurrWin CharSize 2dup 2/ swap 2/ swap 2swap
    currwin ClientSize 2swap d-
    idc_grid ctlhdl WinDims!
;

Dialog: TestDlg
    20 20 200 100 Position&Size
    WS_Visible Style+
    WS_SysMenu Style+
    ws_thickframe style+

    10 100 0 Font "MS Sans Serif"
    2 20 196 60    IDC_Grid        z" "        MyGrid

\    assign textbox-posgrid  to-message WM_SIZE
    WM_SIZE         does textbox-posgrid

\    GCM-RejectChar  does: wParam [char] 9 = .s ReturnVal ! ;
end-dialog
```

To display the dialog box, the word `TST` is defined below. `TST` opens the dialog box and obtains the handle to the grid control saving it in tha value `hGC`. The four columns of the control are

given titles here - that job could have been done by the grid controls GWM_INITCONTROL message handler.

```
0 value hGC

: tst
    0 TestDlg ShowModeless Set-Currwin
    IDC_Grid CtlHdl to hGC
    z" Editable"        hGC 0 gc-SetTitlez
    z" Editable"        hGC 1 gc-SetTitlez
    z" Digits only"     hGC 2 gc-SetTitlez
    z" Not editable"    hGC 3 gc-SetTitlez
;
```

To test this code, enter `TST` and press `Enter`. Use the handle of the grid control to set some of the grid properties with the GC-xxx words.

16 Header Controls

16.1 Introduction

Header controls are used to provide titles above columns of text within other controls. Header controls are split into seperate items (one for each column) each of which has its own properties such as width and text and individual items can be dragged to another position.

Provided here are two GUIgen header controls : *HeaderControl* is a standard header and *ButtonHeader* has the additional style `HDS_BUTTONS` which shows the individual header items as raised button-like objects.

Header Control Styles

The following Window styles can be given to a header control defined by `Another`:

• HDS_HORZ - a default style. Header controls are shown as horizontal. (There is no vertical style).
• HDS_BUTTONS - Displays header items as buttons.
• HDS_HOTTRACK -
• HDS_HIDDEN - hides the control by supplying a value of zero as its height in response to a HDM_LAYOUT message.
• HDS_DRAGDROP -
• HDS_FULLDRAG -

Header Control Messages

Windows provides the following message for header controls. Many of the words in this glossary are wrappers around these messages.

• HDM_DELETEITEM
• HDM_GETITEM
• HDM_GETITEMCOUNT
• HDM_HITTEST
• HDM_INSERTITEM
• HDM_LAYOUT
• HDM_SETITEM

Notification message are sent to a header control's parent window:

• HDN_BEGINTRACK - Signals the start of divider dragging.
• HDN_DIVIDERDBLCLICK - Indicates that the user double-clicked a divider.
• HDN_ENDTRACK - Signals the end of divider dragging.
• HDN_ITEMCHANGED - Indicates a change in the attributes of an item.
• HDN_ITEMCHANGING - Indicates that the attributes of an item are about to change.
• HDN_ITEMCLICK - Indicates that the user clicked an item.
• HDN_ITEMDBLCLICK - Indicates that the user double-clicked an item.
• HDN_TRACK - Indicates that the user dragged a divider.

• HDN_BEGINDRAG
• HDN_ENDDRAG

```
: hcCount       \ hHdr -- n
```
Returns the number of items in the Header Control with handle hHDR.

```
: hcDelete      \ hHdr n --
```
Delete the n-th item of the header control whose Windows handle is hHdr.

```
: hcClear   \ hHdr --
```
Empty the header control - delete all of its items.

```
: Fit-Header-In     \ hHDR x1 y1 x2 y2 --
```
Position a header item within the given rectangle. The height of the rectangle is irrelavent.

```
: hcFitToWin        \ hHdr hWnd --
```
Position a header item within the client area of a given window. The height of the window is irrelavent.

```
: hcFitToParent     \ hHdr --
```
Position a header item within the client area of its parent window.

```
20 Value <hiwidth>
```
The default header item width used when inserting/adding items to a header control. Easily changed by an application if required. as a header item. The new item becomes the idx1-th unless there were fewer than idx1 items initially. idx2 is the new items index or -1 if the operation failed. The new items width is the value of <hiwidth>

```
: hcAddz      \ zaddr hHdr -- idx
```
Append a new header item to the header control whose Windows handle is hHDR the text is the z-string att address 'zaddr'. idx1-th unless there were fewer than idx1 items initially. idx2 is the new items index or -1 if the operation failed. The new items width is the value of <hiwidth>

```
: hcAdd     \ caddr len hHdr -- idx
```
Append a new header item to the header control whose Windows handle is hHDR.

```
: hciSetWidth    \ w hHdr n --
```
Set the n-th item of the header control with handle hHDR to the value w. (w is in pixels)

```
: hciWidth       \ hHdr n -- width
```
Return the width in pixels of the n-th item of the given header control.

```
: hciFormat      \ hHdr n -- fmt
```
Return the text alignment and other format information as a set of bit flags for the given header item.

```
: hciSetFormat   \ hHdr n fmt --
```
Sets the header item formatting value to fmt. Used by the HdrItem-Align_xxx_ words.

```
: hciSetAlign    \ hHdr n val --
```
Set the text alignment for the n-th header item to val.

```
: hciLeft    \ hHdr n --
```
Set the text in the n-th header item to be left aligned.

```
: hciRight   \ hHdr n --
```
Set the text in the n-th header item to be right aligned.

```
: hciCentre  \ hHdr n --
```

Set the text in the n-th header item to be centred .

`: hciCopyText \ hHdr n addr len --`
Copies the text of the given header item to the address 'addr' as a zero terminated string. A maximum of 'len' characters are copied including the terminating zero.

`: hciTextz \ hHdr n -- addrz`
Return a copy of the text of the given header item. Note that the text is copied to a local buffer limited in size to 100 bytes.

`: hciText \ hHDR n -- addr len`
Return a copy of the text of the given header item. Note that the text is copied to a local buffer limited in size to 100 bytes. The string is also zero terminated.

`: hciSetTextz \ zaddr hHdr n --`
Set the text of the n-th header item. zaddr is the address of a zero-terminated string which specifies the text. hHdr is the Windows handle of a header control.

`: hciSetText \ addr len hHdr n --`
Set the text of the n-th header item. addr/len is the address and length of a text string. hHdr is the Windows handle of a header control.

`: hciValue \ hHdr n -- u`
Return the application defined value associated with this header item.

`: hciSetValue \ u hHDR n --`
Assign the application defined value to this header item.

`: hcSetCount \ n hHdr --`
Set the number of items of a header to n. hHdr is the Windows handle of the header control.

`: hcSumWidths \ hHdr -- n`
Return the sum of each item width

`: hcStretchFit \ hHdr --`
Stretch (or shrink) the RH column to fit the width of the control

16.2 Testing/Example

17 TabBook Controls

17.1 Introduction

A *TabBook* is a 'compound control' consisting of a Windows Tab Control plus a number of *pages* each of which is a dialog box usually containing other controls. Because of this compexity, TabControls are defined in a slighly different manner to other GUIgen controls.

All words are compiled into the vocabulary `TabControls` and those for use by applications are `export`ed to Forth.

The following conventions are followed :

• Stack comments and other documentation:

1. *hTBk* is used to indicate the Windows' handle of a TabBook.
2. *caddr ulen* indicates the address and length of a character string.
3. *n* is a general single cell integer - often a page number.

• Word Names:

1. The prefix *tbk* is applied to each 'high level' word. (i.e. those intended for use in application code).
2. Words used within or by a TabBook Windows message handler are indicated by being prefixed by *tc-* and enclosed in angled brackets. (For instance <*tc-HideCurr*>).

17.2 Glossary

17.2.1 Preparation

`vocabulary TabControls`
The vocabulary into which all words are compiled. Words usefull for applications are exported to Forth.

`struct /TCItem  \ -- size ; (TC_ITEM in Windows/C)`
The Windows *TC_ITEM* structure used to transfer information to and from a tab control. Defined as follows:

```
    Cell  field tcitem.mask         // value specifying which members to retrieve or set
    Cell  field tcitem.lpReserved1  // reserved; do not use
    Cell  field tcitem.lpReserved2  // reserved; do not use
    Cell  field tcitem.pszText      // pointer to string containing tab text
    Cell  field tcitem.ccTextMax    // size of buffer pointed to by the pszText member
    Cell  field tcitem.iImage       // index to tab control's image
    Cell  field tcitem.lParam       // application-defined data associated with tab
end-struct
```

`AppMessage: TBM-MatchPage`
A message used to ensure the correct page is shown - that is the page shown matches the current tab index. Not intended to be used in application code.

`AppMessage: TBM-TBINFO`
A message used to implement various information exchange words.

`AppMessage: TBM-PAGECHANGED`
A message sent to a tab page whenever it is shown and whenever it is hidden. wParamLo is its page index. wParamHi is true if the page has been hidden, False if it has been shown. lParam is tab control handle.

17.2.2 TabBook manipulation words

The following words are essentially wrappers around Windows Tab Control messages. They provide a more easily remembered way of obtaining and setting TabBook information.

`: tbkCount  \ hTBk -- n`
Returns the number of pages in a TabBook.

`: tbkPage   \ hTBk -- n`
Returns the current active page of a TabBook.

`: tbkSetPage   \ n hTBk --`
Sets the active page for the TabBook to page number n. This word sends a `TCN_SELCHANGING` notification message to the TabBook's parent first and no action is taken if TRUE is returned. Otherwise the new page number is set and a `TCN_SELCHANGE` notification message sent to the parent informing it of the change. These actions are an attempt to mimic the action of a user clicking on a tab.

`: tbkText   \ n hTBk -- caddr ulen`
Retreive the text of page number *n* for a tabbook. *caddr* is the address of a local buffer.

`: tbkSetText    \ caddr ulen n hTBk --`
Sets the tab text of a TabBook's *n*-th page.

`: tbkNext   \ hTBk --`
Displays the tab page to the right of the currently selected page. If the right-most tab is currently shown the first (left-most) page is displayed.

`: tbkPrev   \ hTBk --`
Displays the tab page to the left of the currently selected page. If the left-most tab is currently shown the last (right-most) page is displayed.

`: tbkPgHdl  \ hTBk n -- hDlg`
Return the dialog box handle of the n-th page of the tab book whose handle in hTBk.

`: tbkCtlHdl \ hTBk n id - hCtl`
Return the handle of the control which appears on the n-th page of the tab book with handle hTBk.

17.2.3 The Tab Page

A TabBook contains a number of pages each of which contains its own controls. Each *TabPage* is a dialog box based on its own window class : `TPageClass`.

A TabPage is created with the word `TabPage:` to make a dialog box dictionary entry. All the standard dialog box 'scripting' words are now available and are used to specify the page caption and all its controls. Finally the definition is terminated using `end-tabpage`.

`DlgClass: TPageClass`

The window class on which TabPages are built.

`: end-tabpage \ --`
A synonym of `end-dialog` used to end a TabPage definition.

`: TabPage: \ "name" ++`
The TabPage defining word. Creates a dialog dictionary entry with its required window and dialog styles.

Example

```
TabPage: ToolsPage
   caption "Tools"
   24 24 62 34 idc_static z" Status"   GroupBox
   30 36 50 14 IDC_Ed2    z" Status"   ViewBox
end-tabpage
```

17.2.4 The TabBook Control

The following words will not normally be used in application code. Those words surrounded by angled brackets (such as `<tc-PageDlg>`) are for use within the TabBook's window procedure only.

`WinProps: /TabBook \ a WinProps structure`
The GUIgen *WinProps* structure which is part of each TabBook control. It is defined as :

```
    cell    field   tb.CtrlKey  \ TRUE if CTRL-key is dowm
    cell    field   tb.hPage    \ handle of current page dialog
    10 cell array-of tb.hDlgs   \ handle of each page...
end-struct
```

`struct /TBControl`
This is an extension to the *WinControl* structure which is `ALLOT`ed as part of a TabBook dictionary entry. The extra space is used to store information about the TabPages which form part of TabBook currently being defined.

`: <tc-PageDlg> \ n -- 'dlg`
Returns the address of the *n*-th TabPage dialog box.

`: <tc-HideCurr> \ -- ;`
Hide the current page dialog.

`: <tc-RevealPg> \ n --`
Show page number *n* of the current TabBook. Any currently shown page is hidden first using `<tc-HideCurr>`.

`: <tc-CreatePg> \ n --`
Executed during TabBook initialisation, this word creates the *n*-th TabPage dialog box, sets it to *invisible* and stores its handle in the *tb.hDlg* field of the TabBook's winprops data.

`: <tc-MatchTab> \ --`
Find which tab is selected and show the corresponding page.

`: <tc-OpenPg> \ n --`

Selects the n-th tab and displays the corresponding page.

`: <tc-CreateTab>  \ n --`
Create TabPage number n. Executed as part of a TabBook's initialisation routine.

`: <tc-RefNotify>  \ --`
The handler for the CN_NOTIFY message which is sent by the TabBook's parent on receipt of a WM_NOTIFY message for this control. This routine performs a single function: on detection of a TCN_SELCHANGE notification message `<tc-RevealPg>` is called - thus ensuring the TabPage displayed matches the tab selected.

`: <tc-SetFont>  \ --`
The WM_SETFONT handler for the TabBook control. Sends a WM_SETFONT message to each TabPage dialog boxes.

17.2.5 Processing the <Tab> key.

Many applications allow users to change pages using the key combinations **Ctrl+Tab** and **Ctrl+Shift+Tab**. The following code which handles WM_KEYDOWN WM_KEYUP and WM_GETDLGCODE messages does just that.

The pressing of a control key is detected and 'remebered' by setting a flag in the tabbook's WinProps. When a control key is released the flag is set to False.

The sneaky bit is in handling the WM_GETDLGCODE message which always returns the default/inherited value. But if the control key flag is set the value DLGC_WANTTAB is additionaly returned. WM_KEYDOWN now detects the tab key when the next tab page is displayed, or the previous page if SHIFT is also pressed.

`: <tc-KeyDown>  \ --`
WM_KEYDOWN message handler. If the key pressed down is a control key, the flag tb.CtrlKey is set to TRUE. If the tab key is pressed the current page is changed.

`: <tc-KeyUp>  \ --`
WM_KEYUP massage handler. Detects the control key being released and resets tb.CtrlKey.

`: <tc-GetDlgCode>  \ --`
WM_GETDLGCODE message handler. Returns the inherited result and in addition returns DLGC_WANTTAB if the control key is pressed.

`WinControl: (TabControl) "SysTabControl32"`
This is the Tab Control. Not for use in application code - use `TabBook:` instead.

`: TabPage,  \ 'dlg --`
Used within a TabBook control definition. *'dlg* is the address of a TabPage definition in the Forth dictionary which is 'embedded' into the TabBook definition. This address is used to create the TabBook when the TabBook is initialied.

`: TabBook:  \ "name" ++`
The word to start a TabBook definition which is ended with `end-control`. Only `TabPage,` will normally appear within this control's definition.
The following Windows styles can be applied to a TabBook control.

a. From WIndows 98 onwards:
   - TCS_BUTTONS - Tabs shown as buttons and no border is shown around the control.
   - TCS_FIXEDWIDTH - All tabs have the same width. (Cannot be used with TCS_RIGHTJUSTIFY)

- TCS_FOCUSNEVER - Tab never receives the input focus.
- TCS_FOCUSONBUTTONDOWN - Tabs receive input focus when clicked.
- TCS_FORCEICONLEFT - Icon forced left on fixed width tabs.
- TCS_FORCELABELLEFT - Labels forced left on fixed width tabs.
- TCS_MULTILINE - displays over multiple lines if necessary.
- TCS_OWNERDRAWFIXED - parent window receives WM_DRAWITEM message to paint tabs. (GUIgen reflects WM_DRAWITEM messages as GWM_DRAWSELF messages by default).
- TCS_RAGGEDRIGHT - default style - tabs are not stretched to fill the entire width of the control.
- TCS_RIGHTJUSTIFY - tabs stretched to fit width if used with the TCS_MULTILINE style.
- TCS_SINGLELINE - the default - "not multiline".
- TCS_TABS - Tabs not shown as buttons - default value.
- TCS_TOOLTIPS - has a tooltip control associated with it.

b. For later versions of Windows:

- TCS_SCROLLOPPOSITE
- TCS_BOTTOM
- TCS_RIGHT
- TCS_MULTISELECT
- TCS_FLATBUTTONS
- TCS_HOTTRACK
- TCS_VERTICAL
- TCS_EX_FLATSEPARATORS
- TCS_EX_REGISTERDROP

17.2.6 Exported words for use in application code

Defining a TabBook

```
export TabPage:
export TabPage,
export TabBook:
```

TabBook manipulation

```
export tbkCount     \ hTBk -- n
export tbkPage      \ hTBk -- n
export tbkSetPage   \ n hTBk --
export tbkText      \ n hTBk -- caddr ulen
export tbkSetText   \ caddr ulen n hTBk --
export tbkNext      \ hTBk --
export tbkPrev      \ hWnd --
export tbkPgHdl     \ hTBk n -- hDlg
export tbkCtlHdl    \ hTBk n id - hCtl
Export TBM-PAGECHANGED  \ -- n
```

17.3 An Example

Defining a simple TabBook :

```
nextid: IDC_Edit
nextID: IDC_Ed2


\ the following code defines three TabBook pages and creates
\ a dictionary entry for each : TPG0 TPG1 and TP2. Each page
\ is a dialog box which contains its own controls. Each page's
\ caption becomes the text shown on the TabBook's tabs.

TabPage: TPg0
    caption "&A Page 0"
    4 4 30 12 idc_static z" Hello" TextLabel
    20 20 60 14 IDC_Edit z" gray"  EditBox
end-dialog


TabPage: TPg1
    caption "&B Page 1"
    24 24 62 34 idc_static z" Status"   GroupBox
    30 36 50 14 IDC_Ed2    z" Status"   ViewBox
end-dialog

TabPage: tp2
    caption "&C ok?"
    24 24 30 12 idc_static z" Page 2"   TextLabel
end-tabpage


\ All these pages come together in a single TabBook: definition:
TabBook: Book1
\    TCS_BUTTONS Style+
    TPg0 TabPage,
    TPg1 TabPage,
    tp2  TabPage,
end-control


\ ..and this TabBook can now be used as any other GuiGen control..

NextID: IDC-Bk
NextID: IDC-Lbl

dialog: db1
    caption " Test box"
    10 10 144 134 Position&Size

    WS_VISIBLE WS_CAPTION or Style
    WS_SYSMENU Style+
```

```
    10 100 0 Font "Arial"

    2 2 120 100     IDC-Bk  zNull           Book1
    10 110 30 12    IDOK    z" &Close"      PushButton

\    WM_NOTIFY   does Reflect-Notify
    WM_COMMAND  does: wParamLo IDOK = if Shut-CurrDlg then ;

end-dialog
```

---

18 Treeview Controls

18.1 Introduction

This document contains a description of the Windows Tree-View control implemented as a GUIgen *Treeview* object.

Like any other GUIgen control, *Treeview* may be used in dialog boxes, standard windows and other controls. It is perhaps best used as a base class for application specific controls (by using the word `Another`) where a Treeview's behaviour can be modified by the addition of extra styles actions.

This wordset was compiled into a vocabulary which as well as serving as an organisational aid provides a simple form of data hiding while allowing 'data revealing' if wanted. 'High level' words - those expected to be used within application code - are *export*ed in a similar way to words exported from a Forth module. All exported words begin with the letters *tv* in order to distinguish them from similar words which act on other controls. (For example `tvClear` empties a Treeview while `LB-Clear` empties a ListBox).

Every word is included in the glossary which follows whether it is a high-level word or not. But for those not interested in the minutiae there follows a glossary of exported words.

An example of a Treeview used in a dialog box is given at the end of this document. Please study this code if you want to learn how to use Treeviews quickly.

18.1.1 Notations

The following conventions have been followed when naming words:

1. Words acting on a Treeview control (not on an individual item) are prefixed by *tv*.
2. Words which act on treeview items are prefixed by *tvi*.

The few exceptions (such as `tvimgFolders`) should cause no confusion.

All words acting on a Treeview require its Window's *handle* which is denoted by *hCtl*. Words targeting a particular Treeview item also require that item's handle which is shown as *hItem*. A list index number is represented by *idx* and character strings by the pair *caddr ulen*. Each of these symbols represent single cell items on the stack.

18.1.2 Setting up and preparation

`vocabulary TreeViews`
The vocabulary into which all words in this wordset are compiled. Application-words are `export`ed from this vocabulary into `Forth`.

`struct /TV_Item`
This Windows data structure is used with many messages passed to and from Windows - adding items, setting and retreiving item text and other properties, etc...

```
_UINT       field tvi.Mask
_HTREEITEM  field tvi.hItem
_UINT       field tvi.State
```

```
    _UINT       field tvi.StateMask
    _LPSTR      field tvi.Text
    _Int        field tvi.TextMax
    _INT        field tvi.Image
    _INT        field tvi.SelectedImage
    _INT        field tvi.Children
    _LPARAM     field tvi.lParam
end-struct
```

`struct /TV_InsertStruct`
The structure is used when adding (inserting) an item to a Treeview.

```
    _HTREEITEM  field tvis.hParent
    _HTREEITEM  field tvis.hInsertAfter
    /TV_Item    field tvis.Item
end-struct
```

`struct /NM_TreeView`
WM_NOTIFY messages from treeview controls are sent with this extended NMHDR structure:

```
    NMHDR +
    cell        field   nmtview.action
    /TV_ITEM    field   nmtview.itemOld
    /TV_ITEM    field   nmtview.itemNew
    POINT       field   nmtview.ptDrag
end-struct
```

`struct /TVHITTESTINFO`
Used with messages asking Windows which item lies at a particular position on scren:

```
    POINT       field tvht.pt
    cell        field tvht.flags
    _HANDLE     field tvht.hItem
end-struct
```

a missing constant ...

`TV_FIRST 17 +   constant TVM_HITTEST`
Used to ask Windows about the positions of items.

18.1.3 Treeview styles

Windows provides a number of Treeview styles which may be specified within an `Another Treeview ... end-control` control definition. These include :

1. TVS_HASBUTTONS Displays plus (+) and minus (-) buttons next to parent items. To include these buttons with items at the root of the tree view, TVS_LINESATROOT must also be specified.
2. TVS_HASLINES Uses lines to show the hierarchy of items.
3. TVS_LINESATROOT Uses lines to link items at the root of the tree-view control. This value is ignored if TVS_HASLINES is not also specified.
4. TVS_EDITLABELS Allows the user to edit the labels of tree-view items.

5. TVS_DISABLEDRAGDROP Prevents the tree-view control from sending TVN_BEGINDRAG notification messages.
6. TVS_SHOWSELALWAYS Causes a selected item to remain selected when the tree-view control loses focus.
   The following styles are defined for later versions of windows. (In fact I believe they are defined for systems using Internet Explorer version 4 or later).
7. TVS_RTLREADING
8. TVS_NOTOOLTIPS
9. TVS_CHECKBOXES
10. TVS_TRACKSELECT
11. TVS_SINGLEEXPAND
12. TVS_INFOTIP
13. TVS_FULLROWSELECT
14. TVS_NOSCROLL
15. TVS_NONEVENHEIGHT

The following words set Treeview-specific styles at run-time. Treeview nodes may be displayed with lines joining siblings and with what Windows calls 'buttons' - a '+' or '-' sign enclosed in a square next to each item containing children.

```
: TVLines+      \ hCtl --
```

Displays the treeview whose handle is hCtl with lines conecting sibling items. (**TVS_HASLINES** is added to the Treeview's style).

```
: TVRLines+     \ hCtl --
```

Displays the treeview whose handle is hCtl with lines conecting sibling items including at the roor level. (Both **TVS_HASLINES** and **TVS_LINESATROOT** are added to the Treeview's style).

```
: TVButtons+    \ hCtl --
```

Displays the treeview whose handle is hCtl with [+] and [-] buttons against those none-root items containing children. (The **TVS_HASBUTTONS** style is added).

```
: TVRButtons+   \ hCtl --
```

Displays the treeview whose handle is hCtl with [+] and [-] buttons against those items containing children including those items at the root. (Both the **TVS_HASBUTTONS** style and the **TVS_LINESATROOT** are added).

```
: TVLines-      \ hCtl --
```

Removes the lines and buttons styles assigned by the preceding words. (**TVS_HASLINES**, **TVS_LINESATROOT** and **TVS_HASBUTTONS** styles are all removed).

18.1.4 Tree View image lists

Description

A Treeview control can use up to two image lists to display icon-type images on the left of tree view items - the *item image list* and the *state image list*. Windows provides a large range of options for using these image lists including optional overlay masks. This code takes a much simpler approach which is adequate in many circumstances.

Item Images

Use `tvSetItemImages` to set a Treeview's item image list passing the address of a bitmap structure which has been created with `BitmapList:` (See the Bitmaps.fth in GUIgen). Once done each item of the Treeview shows an image from that list to the left of the item text. Use `tviSetImageIdx` to specify the index of the image to be shown against an item when unselected, and `tviSetSelectedIdx` to specify the image index when selected.

Images 16 pixels square seem to be suitable but 24 and 32 pixels may work well on high resolution screens - Windows adjusts the height of each Treeview item to the height of the item images.

State Images

The state image list contains images intended to represent the **application-defined** state of treeview items. States are numbered 1 to 15 which indicate the zero-based index (into the state image list) of the image displayed next to an item. A state of zero indicates no image.

This code defines state values 1 and 2 as being *Expanded* and *Collapsed* (i.e. not expanded). `tviShowCollapsed` sets an item's stsate to 1 and `tviShowExpanded` sets it to 2. Use `Reflect-notify` to return **WM_NOTIFY** messages back to a Treeview control from its parent - this code will process **TVN_ITEMEXPANDED** messages to set item image states to 1 or 2 accordingly.

Three state image lists are included in this code any of which may be used to indicate the *Expanded/Collapsed* state. Use the `GGStyle` control property in the definition of `Another` Treeview control and specify any one of `TVSFolder` `TVSBlueBook` or `TVSRedBook`.

You can use `BitmapList:` to define you own state image list and assign it to a Treeview with `tvSetStateImages`. (A suitable image list is 16 pixels high, 48 pixels wide and contains 3 images each 16 pixels square - image zero is unused).

Create an image list containing more images if you invent more states and execute `tviStateIdx` to set each item accordingly.

If you do not want to use state numbers 1 and 2 to represent the *expanded* and *collapsed* states no not reflect **WM_NOTIFY** messages and process the **TVN_ITEMEXPANDED** notification message to suit.

The Image List words ...

`BitmapList: TVImgFolder "%GUIlib%\TVFolder.bmp"`
A Bitmap used as the state image list for Treeviews with the `TVSFolders` GGStyle containing open and closed folder images. This word is made public so that an application may easily mimic Windows Explorer's behaviour of showing selected items with an open folder while all others are show with a closed one.

`BitmapList: TVImgBlueBook "%GUIlib%\TVBlueBook.bmp"`
The image containing images of a open and closed blue books which is used as the state image list for Treeviews with the `TVSBlueBook` GGStyle.

`BitmapList: TVImgRedBook "%GUIlib%\TVRedBook.bmp"`
The image containing images of a open and closed red books which is used as the state image list for Treeviews with the `TVSRedBook` GGStyle.

GUIgen Treeview Styles

The following values are defined for use with Treeview controls. Use them within Treeview definitions with the `GGStyle+` word. NOTE the values are NOT bit flags!

`1 constant TVSFolder`
The GUIgen style causing a Treeview to set `TVImgFolder` as its state image list and so show an open folder alongside expanded items and a closed folder alongside collapsed ones.

`2 constant TVSBlueBook`
A GUIgen style causing a Treeview to show a blue book either opened or closed against expanded and collapsed items as appropriate which is done by setting the state image list to `TVImgBlueBook` when the control is created.

`3 constant TVSRedBook`
A GUIgen style causing a Treeview to set `TVImgRedBook` as its state image list when created. A red book is then displayed alongside each item - a closed book for collapsed (or empty) items and an opened one for expanded items.

`: tvSetStateImages \ BMList hCtl --`
BMList is the address of a `BitmapList:` bitmap list structure and *hCtl* is the handle of a Treeview control. An image list is created and set as the Treeview's state image list. Any previously used state image list is destroyed.

`: tvSetItemImages  \ BMList hCtl --`
BMList is the address of a `BitmapList:` bitmap list structure and *hCtl* is the handle of a Treeview control. An image list is created and set as the Treeview's item image list. Any previously used state image list is destroyed.

`: tviImageIdx       \ hItem hCtl -- idx`
Returns the item image list index of the Treeview whose handle is *hCtl* and item whose handle is *hItem* as the integer *idx*. If the action failed -1 is returned.

`: tviSetImageIdx    \ idx hItem hCtl --`
Sets the item image list index to *idx* for the item *hItem* belonging to Treeview *hCtl*.

`: tviSelectedIdx    \ hItem hCtl -- idx`
Returns the item image list *selected index* of the Treeview whose handel is *hCtl* and item whose handle is *hItem* as the integer *idx*. If the action failed -1 is returned.

`: tviSetSelectedIdx \ idx hItem hCtl --`
For the Treeview whose handle is *hCtl*, sets the selected index for item *hItem* to the value *idx*.

`: tviStateIdx   \ hItem hCtl -- idx`
Returns the state image index of the item whose handle is *hItem* in Treview whose handle is *hCtl*. If n is negative the operation failed.

`: tviSetStateIdx    \ idx hItem hCtl --`
Sets a Treeview item's state index to *n*. *hItem* is the item's handle and *hCtl* is the Treview's handle.

`: tviShowCollapsed  \ hItem hCtl --`
Sets the state index of the Treeview item to 1 and if a state image list has been set shows that image.

`: tviShowExpanded   \ hItem hWnd --`
Sets the state index of the Treeview item to 2 and if a state image list has been set shows that image.

18.1.5 Managing Treeview Items

Items are added to a Treeview control as a *root* item using `tvAddRoot` or as a *child* of an existing item using `tviAddChild`.

Any item which contains a child item is said to be the child's parent. The child items of a parent may be 'traversed' using `tviFirst`, `tviNext` and `tviPrev`. The first item in a Treeview can be obtained with `tvRoot`. Each Treeview item (except root items) has a parent which can be identified with `tvParent`.

`tvSelected` returns the currently selected item while `tviSelect` sets an item as being selected. To test if an item is selected use `tviSelected?`.

A Treeview can be emptied completely using `tvClear` and a Treeview item may be emptied with `tvClearItem`. Delete an item using `tvDelete`.

`: tvRoot \ hCtl -- hItem`
Return the first root item of a tree view control.

`: tviFirst \ hItem hCtl -- hChild`
Returns the first child of the given treeview item.

`: tviNext \ hItem1 hCtl -- hItem2`
Return the next sibling item.

`: tviPrev \ hItem1 hCtl -- hItem2`
Return the previous sibling item

`: tvSelected \ hCtl -- hItem`
Return the currently selected item of a tree view control.

`: tviParent \ hItem hCtl -- hTreeItem'`
Returns the given treeview item's parent or null if it has no parent.

`: tviChild \ hItem hCtl n -- hItem2`
Return the n-th child item. If hItem is zero the n-th root item is returned

`: tviDelete \ hItem hCtl --`
Delete a treview item. *hItem* is the handle of a Treeview item, and *hCtl* is the Treeview's handle.

`: tvClear \ hCtl --`
Delete all items in the treeview control.

`: tviClearItem \ hItem hCtl --`
Removes all children from a treeview item. *hItem* is the handle of a Treeview item, and *hCtl* is the Treeview's handle.

`: tviAddChild \ caddr ulen hParentItem hCtl -- hItem`
Add (append) a new treeview child item and return its handle as *hItem*. *hCtl* is a handle of a treeview control, *hParentItem* is the parent treeview item. The new item's *image index* is set to zero, and its *selected index* is set to one - values which may be changed using `TVSetImageIdx` and `TVSetSelectedIdx`.
If the action fails *hItem* is zero (null).

`: tvAddRoot \ caddr ulen hCtl -- hItem`
Add a new root item to the treeview control whose handle is hCtl. hItem is the handle of the new root item or null if the attempt was unsuccessful.

Treeview Item Text

`: tviRcvText \ hItem hCtl caddr ulen -- caddr ulen'`
Receive a Treeview item's text.

`: tviText \ hItem hCtl -- caddr ulen`
Return the text of a given treeview item as a *caddr/len* string which is a local buffer.

`: tviSetText \ caddr ulen hItem hCtl --`
Set the treeview item's text to the string caddr/ulen.

Tree view Item States

From the Win32 HELP file :

Each item in a tree-view control has a current state. For example, an item can be selected, disabled, expanded, and so on. For the most part, the tree-view control automatically sets an item's state to reflect user actions, such as selection of an item. However, you can also set an item's state by using the TVM_SETITEM message and retrieve the current state of an item by using the TVM_GETITEM message.

`: tviStates \ hItem hCtl -- bitflags`
Returns a treeview item's state bits. bitflags contains the state bits which are set. The following bits may be returned:
• TVIS_SELECTED
• TVIS_CUT
• TVIS_DROPHILITED
• TVIS_BOLD
• TVIS_EXPANDED
• TVIS_EXPANDEDONCE
• TVIS_EXPANDPARTIAL

`: tviTestState: \ flag "name" ++ ; hItem hCtl -- u`
A defining word to create item state testing words such as `tviSelected?` below.

`tvis_selected tviTestState: tviSelected?`
A child word of `tviTestState:` which returns TRUE iff a given Treeview item is selected.

`TVIS_EXPANDED tviTestState: tviExpanded?`
A child word of `tviTestState:` which returns TRUE iff a given Treeview item is expanded.
When Windows expands or collapses a Treeview item it firsts sends a `TVN_ITEMEXPANDING` notification message to the parent window. After performing the expansion (or the collapse) a `TVN_ITEMEXPANDED` message is sent. A Parent window can prevent the action by replying to the `TVN_ITEMEXPANDING` message returning a TRUE result.

The message `TVM_EXPAND` used with the `TVE_EXPAND` flag causes an item's list to expand. Used with `TVE_COLLAPSE` the list is collapsed and `TVE_TOGGLE` collapses the list if it is expanded or expands it if collapsed. The documentation states that `TVN_ITEMEXPANDING` and `TVN_ITEMEXPANDED` are not sent when `TVM_EXPAND` is sent. This is **untrue**. On this machine at least which is running Windows 98, both `TVN_ITEMEXPANDING` and `TVN_ITEMEXPANDED` messages are sent to the parent window the first time each item is expanded but not subsequently. In this case it seems that no item may be expanded until FALSE is returned from the `TVN_ITEMEXPANDING` at least once! And afterwards no `TVN_ITEMEXPANDING` are sent.

`: tviExpand \ hItem hCtl --`

Expand a Treeview item's child list by sending a `TVM_EXPAND` window message to the control. Note that contrary to the documentation, Windows *does* send a `TVN_ITEMEXPANDING` notification message in response to this message until the item is expanded after which neither `TVN_ITEMEXPANDING` nor `TVN_ITEMEXPANDED` are sent **to that item**!

`: tviCollapse  \ hItem hCtl --`
Collapses a Treeview items child list if it is expanded.

Miscellaneous

`: tvSetIndent  \ n hCtl --`
Set the width of indentation for a Treeview.

`: tvIndent  \ hCtl -- n`
Returns a Treeview's indentation width.

`: tviChildren?  \ hItem hCtl -- flag`
Returns TRUE iff the item has any children.

`: tvItemAt?  \ x y hCtl -- hItem|0`
Given the coordinates x and y within the client area of the Treeview control whose handle is *hCtl* the handle of the item at that position is returned. If no item lies there zero is returned.

`: tviSelect  \ hItem hCtl --`
Select/highlight the Treeview item specified by *hItem* and *hCtl*

`: tviParam  \ hItem hCtl -- u`
Get the 'lParam' value associated with a treeview item. hCtl is a handle to a tree-view control, and hItem is the handle to the item.

`: tviSetParam  \ u hItem hCtl --`
Set the 'lParam' value associated with a treeview item. hCtl is a handle to a treeview control, and hItem is the handle to the item.

18.1.6 The TreeView control definition

`WinControl: TreeView "SysTreeView32"`
A TreeView control. Defined with 'standard' styles.

`: TVproc-NOTIFY  \ -- ;`
The handler for CM_NOTIFY messages - i.e WM_NOTIFY + CM_MSG0, sent back to the Treeview by its parent in respnse to a WM_NOTIFY message.

`: tvproc-Init  \ --`
The Treeview GWM_InitControl message handler. This word sets the state image list according to the GGStyle value.

18.2 Exported/Public words

The GUIgen Treeview control

```
export TreeView         \ -- addr
```

Image Lists

```
export tvSetItemImages      \ BMF hCtl --
export tviImageIdx          \ hItem hCtl -- idx
export tviSetImageIdx       \ idx hItem hCtl --
export tvSetStateImages     \ BMF hCtl --
export tviSelectedIdx       \ hItem hCtl -- idx
```

```
export tviSetSelectedIdx    \ idx hItem hCtl --
export tviStateIdx          \ hItem hCtl -- idx
export tviSetStateIdx       \ idx hItem hCtl --
export tviShowCollapsed     \ hItem hCtl --
export tviShowExpanded      \ hItem hWnd --
```

State Image Lists

```
export TVImgFolder      \ -- addr
export TVImgBlueBook    \ -- addr
export TVImgRedBook     \ -- addr
```

State Styles

```
export TVSFolder        \ -- u
export TVSBlueBook      \ -- u
export TVSRedBook       \ -- u
```

Adding items

```
export tvAddRoot        \ caddr ulen hCtl -- hItem
export tviAddChild      \ caddr ulen hItem hCtl -- hChild
```

Traversing a Treeview

```
export tvRoot           \ hCtl -- hItem
export tviNext          \ hItem hCtl -- hItem2
export tviPrev          \ hItem hCtl -- hItem2
export tvSelected       \ hCtl -- hItem
export tviFirst         \ hItem hCtl -- hItem2
export tviParent        \ hItem hCtl -- hItem2
export tviChild         \ hItem hCtl n -- hItem2
export tviDelete        \ hItem hCtl --
export tvClear          \ hCtl --
export tviClearItem     \ hItem hCtl --
export tviStates        \ hItem hCtl -- bitflags
```

Item Text

```
export tviRcvText       \ hItem hCtl caddr ulen -- caddr ulen'
export tviText          \ hItem hCtl -- caddr ulen
export tviSetText       \ caddr ulen hItem hCtl --
```

Lines and buttons

```
export TVLines+         \ hCtl --
export TVRLines+        \ hCtl --
export TVButtons+       \ hCtl --
export TVRButtons+      \ hCtl --
export TVLines-         \ hCtl --
```

Miscellaneous

```
export tvSetIndent      \ n hCtl --
export tvIndent         \ hCtl -- n
export tviChildren?     \ hItem hCtl -- flag
```

```
export tvItemAt?        \ x y hCtl -- hItem|0
export tviSelect        \ hItem hCtl --
export tviParam         \ hItem hCtl -- u
export tviSetParam      \ u hItem hCtl --
export tviSelected?     \ hItem hCtl -- flag
export tviExpanded?     \ hItem hCtl -- flag
export tviExpand        \ hItem hCtl --
export tviCollapse      \ hItem hCtl --
export tviTextRect      \ hItem hCtl -- rect
```

Windows Structures

```
export /TV_Item
export tvi.Mask
export tvi.hItem
export tvi.State
export tvi.StateMask
export tvi.Text
export tvi.TextMax
export tvi.Image
export tvi.SelectedImage
export tvi.Children
export tvi.lParam

export /NM_TreeView     \ -- u
export nmtview.action
export nmtview.itemOld
export nmtview.itemNew
export nmtview.ptDrag
```

18.3 An Example

```
Another TreeView MyTreeView
    TVSFolder GGStyle+          \ adds open/closed folder images

    WS_EX_CLIENTEDGE ExStyle+   \ adds sunken border
\    WS_EX_DLGMODALFRAME ExStyle+ \ adds raised border

end-control


NextID: IDC_TView

dialog: TVDlg
    10 10 250 130 Position&Size
    WS_VISIBLE WS_CAPTION or WS_SYSMENU or WS_THICKFRAME or Style
    DS_MODALFRAME DS_3DLOOK or DS_SetFont or Style+
    9 100 0 Font "Arial"
    2 2 150 120     IDC_TView     z" Tree View"     MyTreeView
    205 20 40 15  IDCLOSE       z" &Close"          DefPushButton
end-dialogbox
```

```
0 value TView
0 value Root0
0 value Root1


: TVDlg-Init    \ --

    IDC_TView CtlHdl -> TView

    s" Root Item 0" TView tvAddRoot -> Root0
    s" Root Item 1" TView tvAddRoot -> Root1

    s" Alpha" Root0 TView tviAddChild drop
    s" Beta" Root0 TView tviAddChild drop
    s" Gamma" Root0 TView tviAddChild >r

    s" Ay" r@ TView tviAddChild drop
    s" Bee" r@ TView tviAddChild drop
    s" Cee" r@ TView tviAddChild drop
    s" Dee is a very long wun" r@ TView tviAddChild drop
    s" eey" r@ TView tviAddChild drop
    s" eff" r@ TView tviAddChild drop
    s" gee" r@ TView tviAddChild drop
    s" aitch" r@ TView tviAddChild drop

    r> drop
;


TVDlg [messages
    WM_COMMAND      does:   CurrWin SendClose ;
    WM_INITDIALOG   does    TVDlg-Init
\    WM_NOTIFY       does    Reflect-notify
messages]

: tst
    0 TVDlg ShowModeless set-currwin
;
```

19 The BaseGraph Control

19.1 Introduction

This code provides a control which can be used to display data graphically as lines and points or as histogram bars. Positive X and Y values only are catered for and each axis is scalable and may be labelled.

Typically an application will set the size of the control, set the logical width and height of the graphing area, label each axis and then send the graphical data to be plotted. The graphical data are saved by the control and used to repaint it when required.

Graphical data are sent to the control by words which require the Windows handle of the control and either one or two parameters.

Among the many functions provided are:

1. Set the current position.
2. Plot a line from to a specified position.
3. Plot a point at a position
4. Set the pen plotting colour at a position
5. Set a histograms bar width
6. Add a value to a histogram

See the list of `exported` words at the end of this document for a full list.

Notation

Most words follow the following convention:

1. Exported words are prefixed by *bg*; hence `bgLine`.
2. Words executed within the BaseGraph control window procedure are prefixed by *bgc-*. For example `BGC-PAINT` the WM_PAINT message handler.
3. Window messages begin with *BGM-*.
4. Data structure words begin */BG*.

A few exceptions occur such as `py>ly` where a prefix might otherwise confuse or obscure a words meaning.

19.2 Design Decisions

Approach

A graph control can operate in one of two ways it seems to me. Firstly the lines and points plotted can be saved by the control and used to repaint itself. Alternatively, the application can plot the graph in answer to some message (presumably inside a WM_PAINT message). The second method simplifies the design of the control but arguably makes the application code more complex. A major purpose of using controls is to simplify application code and for that reason this control will take the former approach.

A control will have to maintain a list of points plotted and lines drawn - a history list of some sort. This code uses a linear list of fixed length records. The first cell of each record is used to indicate the type (implying size/length) of data which follows. This has a number of advantages: it is reasonably simple and allows more data types to be added easily.

Drawing Graphs

Windows gives the programmer the ability to choose the logical units he will use to draw things. For instance, it can be specified that the width of a window be 100 units and its height be 150 units say. No matter what the actual size of the window, the graph will be shown appropriately scaled.

Unfortunately, this scaling also applies to text fonts. In this case, if the size of a window is increased the text drawn in that window also increases in size. An effect is not desirable in this case.

So to draw a graph with text labelling of its axes, there are two possible approaches:

a. Perform all scaling within the application code and allow text sizes to remain contant, or
b. Set the x- and y- scales appropriately to let Windows do the scaling, but calculate and set text (font) sizes according to the (y-axis) scale set.

After some experimentation, the option (b) has been abandoned in favour of (a) for the following reasons.

In principle the size of text in case (b) can be calculated to compensate for the different scales. However, at certain sizes of the control, text size appears to 'jump' to a different size as the control's size is changed. Presumably this is due to rounding errors in this code and Windows.

For this reason the scaling is performed by this code.

19.3 Glossary

19.3.1 Setting up and Miscellaneous functions

`vocabulary BaseGraphs`
The vocabulary into which all words are compiled. Application words are later exported into the main Forth vocabulary.

`: w>s  \ n1 -- n2 ;`
Convert a 16-bit "word" to a (32-bit) cell

`: ?/  \ n1 n2 -- n3`
A 'guarded' divide returning zero if n2 is zero otherwise n3 is the quotient of n1 divided by n3.

`Create <DefColours>`
An array of RGB colour values used when colouring a histogram, defined simply as :

```
$FF8080 ,    \ PURPLEY
$FF7FFF ,    \ LT PURPLE
$C0FFFF ,    \ CREAM
$80FFA0 ,    \ PASTEL GREEN
$40C000 ,    \ GREEN
```

```
    $8080FF ,    \ DUSKY PINK
    $FF8000 ,    \ PASTEL BLUE
    $00C0C0 ,    \ YELLOWY
    $0080FF ,    \ ORANGE
    $0000FF ,    \ RED

: BarColour     \ n -- addr
```

Returns the n-th colour value address from the `BGColours` array.

`10 cells buffer: <UserColours>`

An array available for an application to fill with other colours in which case the word BGColours should be assigned to this action.

19.3.2 Comunication between an application and the BaseGraph Control

A number of messages are used to send data to and receive data from an application. These are defined below along with any data structures used with those messages. An application need not send these messages explicitley since words are provided which do that.

One message is sent by the BaseGraph control asking the application for information: BGM-SpellValue. These two messages are sent to itself if the control has the `BGS-ASKSELF` style. In this way a 'sub-classed' BaseGraph (using `Another`) is able handle its own data entirely on its own providing a way to create more "self-contained" controls.

`1 constant BGS-ASKSELF`

BaseGraph controls with the BGS-ASKSELF style send BGM-SpellValue messages to itself.

Spelling Values

A BaseGraph control will attempt to mark each axis at a number of points and label those marks with their numeric value. By default the value is expressed as a decimal integer, but by responding to the `BGM-SpellValue` message an application can supply the control with different text - perhaps by interpreting the value as a scaled integer.

`AppMessage: BGM-SpellValue`

A message sent to the control's parent window to request the text representation of a value. wParam is the handle of the control and lParam is a `/BGSpell` structure. The handler for this message should return none-zero to indicate it has been handled. If the return value is zero the value is converted to text as a decimal integer.

`/BGSpell` is the name of the data structure sent with a `BGM-SpellValue` message consisting of three fields as follows:

```
struct /BGSpell
    cell        field   bgs.Value       \ the value to be spelt
    cell        field   bgs.Abscissa    \ true if for the x-axis
    24 chars    field   bgs.Text        \ counted string
end-struct
```

The BGSpell structure is sent to the BaseGraph control parent (or itself if it has the `BGS-ASKSELF` style), whenever the text representation of a y- or x-value is required, as the *lParam* parameter of a `BGM-SpellValue` message.

bgs.Value holds the value which is to be converted to text and placed as a counted string in the field *bgs.Text*. *bgs.Abcessa* is zero (false) if the value refers to the y-axis and non-zero (true) if the x-axis. An application will return a non-zero value to indicate that *bgs.Text* contains valid text.

If a zero value is returned from sending a `BGM-SpellValue` message a default routine converts the value to a decimal value with a number of decimal places as held in the *bgc.DPX* or *bgc.DPY* fields of the BaseGraph *WinProps* data. (See the `/BGClass` structure and the words `bgSet-XDP` and `bgSet-YDP` below).

The Plot Commands

An application plots data by sending the BaseGraph control a number of `BGM-Plot` messages in the guise of bgXXX words (defined below). The following plot operations are provided:

- Set current position
- Draw a line from current position to another position.
- Plot a point
- Plot a 'circle point'.
- plot a 'square point'.
- plot a 'diamond point'.
- plot a 'cross point' (similar to 'x')
- plot a 'cross point'. (similar to '+')
- Set the pen line colour
- Set the pen thickness
- Set the point shape 'radius'

`AppMessage: BGM-Plot`
The message used to send a 'plotting operation' to the control. wParam is a /BGOP structure lParam is currently unused.

```
struct /bgop    \ Basic Graph OPeration
```
The structure used to record each drawing operation sent to the control.

```
: SendPlot      \ x y hCtl index --
```
The word which sends `BGM-Plot` message to a control. This is a factorisation of `Plotter:` and other words.

x and *y* are cell-sized values often indicating logical x and logical y values but are also used for different things such a colour value.

hCtl is the hadle of the control and *index* is the operation identifier.

```
: Plotter:      \ index "name" ++ (coimpiling); u1 u2 hCtl -- (child)
```
A defining word used to create Plot Command words which take two parameters (x and y values).
The following Plot Commands are created using Plotter: with each x and y value being in logical corrdinates.

```
0 Plotter: bgAt       \ x y hBG --
```
Move the current position to (x, y).

`1 Plotter: bgLine \ x y hBG --`
Draw a line from the current position to (x, y).

`2 Plotter: bgPoint \ x y hBG --`
Draw a point in the shape of a solid circle (or dot) with the current point size at logical position (x, y).

`3 Plotter: bgCircle \ x y hBG --`
Draw a point in the shape of a circle with the current point size at logical position (x, y).

`4 Plotter: bgSquare \ x y hBG --`
Draw a point in the shape of a square with the current point size at logical position (x, y).

`5 Plotter: bgCross \ x y hBG --`
Draw a point in the shape of a "X" with the current point size, at logical position (x, y).

`6 Plotter: bgCross2 \ x y hBG --`
Draw a point in the shape of a "+" with the current point size, at logical position (x, y).

`7 Plotter: bgDiamond \ x y hBG --`
Draw a point in the shape of a diamond with the current point size, at logical position (x, y).

`: bgPenColour \ colour hBG --`
Set the current plotting colour (the pen colour) to that given.

`: bgLineWidth \ width hBG --`
Set the current line width to 'width'.

`: bgPointsize \ radius hBG --`
Set the current point size to the value *radius*.

Getting and Setting BaseGraph Values

Two messages are used to get and set a BaseGraph Contol's values. These messages are not intended for use by applications since 'wrapper' words have been provided.

`AppMessage: BGM-SetValue`
The message sent to a Base Graph control to set various values. 'Wrapper' words are provided in the form bgSet-xxxx.

wParam holds the value to be set. lParamlo indicates the type of value being set. lParamHi may hold extra information depending on the value being set.

The following values are currently defined (first number is the lParamLo index, the value being set is in wParam):

- $00 Set the maximum value of X
- $01 Set the maximum value of Y
- $02 Set the number of decimal places of each X value
- $03 Set the number of decimal places of each Y value
- $04 Set the width of all bars
- $05 Set the graph mode.
- $06 Set the number of bars to show
- $07 Set the number of cartesian number x-axis divisions
- $08 Set the x-axis label. wParam= address, length=lParamHi

- $09 Set the y-axis label. wParam= address, length=lParamHi
- $10 Add a value to the graph. The bar which the value falls in is incremented by one.
- $11 Set the value of a particular bar. Index of bar is in lParamHi
- $12 Add another bar and set its value to wParam
- $13 Set a bar label; wParam=address of counted string, lParamHi is the bar number.
- $14 Set a the lParamHi-th bar colour to wParam
- $15 Set normal distibution parameters.
- $20 (lParamHi=0) Clear histogram data
- $20 (lParamHi=1) Clear points data
- $20 (lParamHi=2) Show all data bars with maximum Y a power of 10
- $20 (lParamHi=3) like wParamHo=2 but Max Y may be 3 times power of 10
- $20 (lParamHi=4) Fit the height so all bars shown
- $20 (lParamHi=5) Increase maximum value of X, if necessary to make all bars fit.
- $20 (lParamHi=$0A) Show the histogram cumulative data line.
- $20 (lParamHi=$0B) Hide the histogram cumulative data line.
- $20 (lParamHi=$0C) Set the POI value
- $20 (lParamHi=$0D) Set the TooMany flag
- $20 (lParamHi=$0E) Set the Plot Percents flag
- $20 (lParamHi=$0F) Show/Hide the histogram grid lines.
- $20 (lParamHi=$10) Set grid colour
- $20 (lParamHi=$11) Show/Hide the Mean line.
- $20 (lParamHi=$12) Set mean colour
- $20 (lParamHi=$13) Set the mean value
- $20 (lParamHi=$14) Show/Hide the normal distribution
- $20 (lParamHi=$15) Set norm. dist. colour
- $20 (lParamHi=$16) Adjust and set the bar count to fit
- $20 (lParamHi=$17) Clear all bar colours.

`AppMessage: BGM-GetValue`

A messages sent to a control to obtain various values which can be set using BGM-SetValue. The index number in lParamLo indicates which value is being sought. lParamHi and wParam vary. An application should use the *bgcSet-xxx* words rather than using this message explicitly. The following index numbers are defined:

- $00 Return the Maximum X value
- $01 Return the maximum Y value
- $02 Return the decimal points of X values
- $03 Return the decimal points of Y values
- $04 Return width of bars
- $05 Return the graph mode
- $06 Return the number of bars
- $07 Resturn the number of x-axis numbered divisions
- $08 Return x-axis label
- $09 Return y-axis label
- $10 Unused
- $11 Return value of the nth bar, with n being the value of lParamHi
- $13 Return the nth bar label, with n being the value of lParamHi is the bar number.
- $14 Return the n-th bar colour with n being the value of lParamHi
- $15 Get norm. distrib. parameters
- $20 (lParamHi=$00) Get Maximum Frequency
- $20 (lParamHi=$0C) Get the POI value
- $20 (lParamHi=$0D) Get the TooMany flag

- $20 (lParamHi=$0E) Get the 'PlotPercents' flag
- $20 (lParamHi=$0F) Return TRUE if grid lines being shown
- $20 (lParamHi=$10) Return grid colour
- $20 (lParamHi=$11) Return TRUE if Mean line is shown
- $20 (lParamHi=$12) Return mean line colour
- $20 (lParamHi=$13) Return mean value
- $20 (lParamHi=$14) Return TRUE if the normal distribution is shown
- $20 (lParamHi=$15) Return norm dist colour
- $20 (lParamHi=$16) – unused
- $20 (lParamHi=$17) – Unused

The followng words are arranged in 'pairs of pairs' to set and get a particular type of value using the BGM-SetValue message. The value of lParamLo indicates which value is being set.

`$00 ValSetter: bgSet-MaxX \ n hCtl --`
Set the maximum logical value of x which can shown

`$01 ValSetter: bgSet-MaxY \ n hCtl --`
Set the maximum logical value of y which can shown

`$02 ValSetter: bgSet-XDP \ n hCtl --`
Set the number of decimal places that the logical values along the x-axis which are shown when marking it.

`$03 ValSetter: bgSet-YDP \ n hCtl --`
Set the number of decimal places that the logical values along the y-axis which are shown when marking it.

`$04 ValSetter: bgSet-BarWidth \ n hCtl --`
Set the width of each hisogram bar to *n* logical units. The bars then cover the intervals [0..n-1], [n..2n-1], [2n..3n-1] ... The number of bars shown is equal to the maximum x value divided by the bar width.

`$05 ValSetter: bgSet-Mode \ n hCtl --`
Set the braph mode to n. This word is not public - wrapper words are defined later.

`$06 ValSetter: bgSet-BarCount \ n hCtl --`
Set the number of bars to n

`$07 ValSetter: bgSet-XDivs \ n hCtl --`
Set the number of x-axis divisions to n

`$10 ValSetter: bgAdd-Value \ n hCtl --`
Add a value *n* (an x-value) to the histogram data. The height of the appropriate bar is then increased by one.

`: bgAdd-Bar \ n hCtl -- index|-1`
Add a new bar to the histogram and set its value to that given. The value returned is the index of the bar added or -1 if the operation failed
The following words also send a BGM-SetValue message to perform a particular action without any other parameters:

`: bgClearHisto \ hCtl --`
Clear the histogram data. That is each bar is set to zero height.

`: bgClearPoints \ hCtl --`
Clear the line & point plot operation list.

`: bgNormHisto \ hCtl --`
'Normalise' the histogram. That is, adjust the logical height of the graph so that no bar has a frequency/value greater than it. In addition the height of the graph (maximum Y value) will be set to a power of ten.

`: bgNormHisto(3) \ hCtl --`
'Normalise' the histogram. That is, adjust the logical height of the graph so that no bar has a frequency/value greater than it. In addition the height of the graph (maximum Y value) will be set to either a power of ten or to 3 times a power of ten.

`: bgFitHeight \ hCtl --`
Adjust the logical height of the histo gram so that all the histogram bars are visible. The resulting logical height is set to the lowest multiple of ten which is equal or greater than the maximum bar value.

`: bgFitWidth \ hCtl --`
Adjust the logical height of the histo gram so that all the histogram bars are visible. The resulting logical height is set to the lowest multiple of ten which is equal or greater than the maximum bar value.

`: bgShowCum \ hCtl --`
Remove all plot points and create a set which gives a cumulative line of the histogram data.

`: bgHideCum \ hCtl --`
Remove all plot points and create a set which gives a cumulative line of the histogram data.
These words use the BGM-SetValue to send *caddr/len* string specifier pairs as parameters:

`: bgSet-XLabel \ caddr ulen hCtl --`
Set the x-axis label to the given text.

`: bgSet-YLabel \ caddr ulen hCtl --`
Set the y-axis label to the given text.

`: bgGet-XLabel \ hCtl -- caddr ulen`
Return the x-axis label text. *caddr/ulen* specifies a character string in a local buffer.

`: bgGet-YLabel \ hCtl -- caddr ulen`
Return the y-axis label text. *caddr/ulen* specifies a character string in a local buffer.

`: bgGet-BarLbl \ hCtl n -- caddr ulen`
Return the n-th bar label text. *caddr/ulen* specifies a character string in a local buffer.

`$00 ValGetter: bgGet-MaxX \ n hCtl --`
Get the maximum logical value of y which can shown

`$01 ValGetter: bgGet-MaxY \ n hCtl --`
Get the maximum logical value of y which can shown

`$02 ValGetter: bgGet-XDP \ n hCtl --`
Set the number of decimal places that the logical values along the x-axis are scaled by.

`$03 ValGetter: bgGet-YDP \ n hCtl --`
Set the number of decimal places that the logical values along the y-axis are scaled by.

`$04 ValGetter: bgGet-BarWidth \ hCtl -- n`
Return the bar width in logical units.

`$05 ValGetter: bgGet-Mode \ hCtl -- n`
Return the current graph mode.

```
$06 ValGetter: bgGet-BarCount \ hCtl -- n
```
Returns the number of bars in the histogram.

```
$07 ValGetter: bgGet-XDivs      \ hCtl -- n
```
Returns the number of x-axis divisions.

19.3.3 The BaseGraph Control

The BaseGraph control is a WinClass-derived control (defined using `AppControl:`). It has a rather large `WinProps` used to save various data which can be divided into three parts: General values which are used when positioning the graph axes, drawing the axis labels and converting logical values into pixel values. Secondly data used only when dealing with histograms, and finally those data used for the line and plotting data.

The control can be in one of three modes:

• Labelled Histogram: a histogram in which the x-axis is/can be marked with a text label beneath each bar.
• Numbered Histogram: a histogram in which the x-axis is marked with nueric values which may not coinside with the histogram bars.
• Point Plotter: a catersian graph on which points and lines are shown.

```
200 constant MaxBars
```
The maximum number of bars in a histogram.

```
winprops: /BGClass
    \ General fields ...
    cell        field   bgc.xoff    \ x-offset to y-axis
    cell        field   bgc.yoff    \ y-offset to x-axis
    cell        field   bgc.mx      \ margin x, at rhs
    cell        field   bgc.my      \ margin y, at top
    cell        field   bgc.MaxX    \ Maximum x value
    cell        field   bgc.MaxY    \ Maximum y value
    cell        field   bgc.xpix    \ x 'extent' of Max X
    cell        field   bgc.ypix    \ y 'extent' of Max Y
    /BGSpell    field   bgc.BGSpell \ receives string version of a value
    64          field   bgc.XLabel  \ x-axis text label
    64          field   bgc.YLabel  \ y-axis text label
    cell        field   bgc.DPX     \ decimal places along x-axis
    cell        field   bgc.DPY     \ decimal places along y-axis
    cell        field   bgc.Mode    \ histogram or points etc..
    \ Histogram specific fields ...
    cell        field   bgc.BarCount    \ number of bars in set
    cell        field   bgc.TooMany     \ attempt was made to set count too high
    cell        field   bgc.PlotPC      \ TRUE iff y-axis show as percentage
    cell        field   bgc.ShowGrid    \ TRUE iff grid lines shown
    cell        field   bgc.GridClr     \ Colour of grid
    cell        field   bgc.ShowMean    \ TRUE iff mean line is shown
    cell        field   bgc.MeanClr     \ Colour of mean
    cell        field   bgc.MeanVal     \ value of mean
    MaxBars cell array-of bgc.Bars      \ count/frequency of each bar
    MaxBars cell array-of bgc.Clrs      \ colour of each bar
```

```
    cell        field   bgc.BarWidth    \ width of each bar (logical units)
    cell        field   bgc.Total       \ Total. Sum of bars.
    MaxBars 24 array-of bgc.BarLbls     \ label strings
    cell        field   bgc.ShowCum     \ True = show histo cumulative line
    cell        field   bgc.POI         \ point of interest (times 10)
    \ Plot commands ...
    cell        field   bgc.Ops.Count   \ total number of operations
    cell        field   bgc.Ops.Used    \ number of operations used
    cell        field   bgc.Ops.Mem     \ holds address of list
    cell        field   bgc.PointSize   \ size of next point plotted
    cell        field   bgc.XDivs       \ number of x scale divisions
    [defined] FPCell [if]
    FPCell      field   bgc.Variance    \ variance of norm dist
    FPCell      field   bgc.-2V         \ used when drawing curve
    FPCell      field   bgc.NDMean      \ mean as float
    cell        field   bgc.ShowND      \ true to display norm dist
    cell        field   bgc.NDClr       \ colour of norm distrib
    FPcell      field   bgc.NDHeight    \ height of norm dist
    [then]
end-struct
```

The window class `BGClass` is defined below. Note that despite such a simple definition, all BaseGraph control message handling is performed by this class. Those actions are defined later and assigned to the class using `[ClassMessages` and `Messages]`.

```
WinClass: BGClass
    /BGClass PropertySize
\    Name "GUIgen-BaseGraphClass"
    CS_HREDRAW CS_VREDRAW or    Style
    CS_DBLCLKS                  Style+

end-winclass
```

The control itself is also simply defined:

```
BGClass AppControl: BGraphControl
    WS_CHILD WS_BORDER or       Style
    WS_VISIBLE                  Style+

    clr-Window colour
end-control
```

19.3.4 Main word Glossary

The drawing operations

```
: lx>px     \ logicalX -- PixelX
```

Return the x-direction pixel value given a logical x value.

```
: ly>py     \ logicalY -- PixelY
```

Return the y-direction pixel value given a logical y value.

`: px>lx \ x1 -- x2 ;`
Change from client pixels *x1* to logical x units *x2*.

`: py>ly \ y1 -- y2 ;`
change from pixels *y1* to logical y units *y2*.

`: op-xy \ op -- pixX pixY`
Given the address of a `/bgop` data structure, return the x and y logical values converted from pixels.

`: bgc-SquarePt \ op --`
Plot a 'square point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-CirclePt \ op --`
Plot a 'circle point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-BlobPt \ op --`
Plot a 'solid point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-Cross \ op --`
Plot a 'X point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-Cross2 \ op --`
Plot a '+ point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-diamond \ op --`
Plot a 'diampnd point' at the position indicated by the given `/bgop` data structure the dimensions determined by the current point size.

`: bgc-SetColour \ op --`
Set the plotting colour to the value held in the `op.x` field of the given `/bgop` structure. Any lines and points ploted from now on will be of this colour.

`: bgc-SetLWidth \ op --`
Set the line to the value held in the `op.x` field of the given `/bgop` structure.

`: bgc-SetPointSize \ op --`
Set the point size to the value held in the `op.x` field of the given `/bgop` structure.

Managing the operation list.

The BaseGraph control maintains a list of plotting operations which it uses to paint the window. The default size of 100 operations will be increased as and when required.

`: bgc-AllocOps \ n --`
Allocate external OS memory enough to store *n* operations. Any previously allocated memory if first `free`d and the number of operations in the list is set to zero.

`: bgc-MoreOps \ -- ;`
Increase the size of the operation list by 100 operations.

`: bgc-TopUp \ -- ;`
Create more operations if list is full.

`: bgc-Op \ n -- 'op`
Return the address of the *n*-th operation.

`: bgc-NewOp \ -- addr ;`
Add new operation to the list calling `bgc-TopUp` to increase the operation list size if neccessary.

`: bgc-DrawOp \ op --`
Performs the drawing operation indicated by the given `/bgop` operation strucure. Used as part of the WM_PAINT message handler.

`: bgc-DrawOps \ -- ;`
Used by the WM_PAINT hadler to draw each of the drawing operations in the operation list.

`: bgc-InitPoints \ -- ;`
The *clear all points* action: all drawing operations are deleted and the plotting defaults are restored.

`: bgc-InitBars \ --`
Initialise or clears the histogram data.

Handling the 'Spell Value' action

`: AskWhom? \ -- hWin`
Returns the BaseGraph controls' parent window handle or, if the control has the BGS-ASKSELF style, returns its own handle.

`: bgc-SpeltForUs? \ n abscissa? -- flag`
Send a BGM-SpellValue to the current window's parent. *n* is the value to convert to text, *abscissa?* is TRUE if *n* represents an x value. *flag* is the value returned from sending the message.

`: bgc-SpellSelf \ n abscissa? -- caddr ulen ;`
Return the text representation of the value *n* for an x value if *abscissa?* is TRUE or for a y value otherwise. This word treats values as scaled decimal numbers with the number of decimal points held in the WinProp fields *bgc.DPX* and *bgc.DPY*. (See also the words `bgc-SetDPY` and `bgc-SetDPY`).

`: bgc-n>text \ n abscissa? -- caddr ulen ; y 0 -- ca u; x 1 -- ca u`
Returns the text representation of the x-value *n* (if *abscissa?* is TRUE) or the y-value *n* by first sending a BGM-SpellValue (using `bgc-SpeltForUs?`) or if that fails using `bgc-SpellSelf`.

`: bgc-x>text \ x -- caddr ulen`
Return the text representaion of the x value given

`: bgc-y>text \ y -- caddr ulen`
Return the text representaion of the y value given

`: bgc-Calc-XMarg \ -- ;`
Use current DC to calculate left-hand and right-hand X-margins storing the results in the appropriate WinProps fields.

`: bgc-Calc-YMarg \ -- ;`
Use current DC to calculate left-hand and right-hand Y-margins storing the results in the appropriate WinProps fields.

`: bgc-CalcScale \ --`
calculate the current scale of the graph by setting the WinProps field *bgc.xpix* and *bgc.ypix* appropriatly. Part of the WM_PAINT handler.

`: bgc-XAxis \ --`
Draw the horizontal x-axis line.

`: bgc-MarkX \ x --`
Place a mark on the x-axis and label it with its value according to `bgc-n>text`.

`: bgc-MarkY \ logicalY --`
Place a mark on the x-axis and label it with its value according to `bgc-n>text`.

`: bgc-YAxis \ --`
Place a number of marks along the y-axis labelling each one

`: bgc-DrawXLabel \ --`
Draw the x-axis text label if one has been specified..

`: bgc-DrawYLabel \ --`
Draw the y-axis text label if one has been specified..

`: bgc-DrawAxes \ -- ; draw the axes which`
Draw both axes and their text labels.

`: bgc-BarLeft \ idx -- pixelX`
Return the horizontal position in pixels of the *idx*-th histogram bar.

`: bgc-BarTop \ idx -- pixelY`
Return the vertical position in pixels of the *idx*-th histogram bar.

`: bgc-DrawBar \ idx --`
Draw the *idx*-th histogram bar, colouring it in with the *idx*-th `BarColour` and outlining it in black.

`: bgc-DrawBars \ --`
Draw all histogram bars.

`: %y>py \ percentageY -- PixelY`
Return the y-direction pixel value given a percentage y value.

`: fly>py \ -- PixelY ; F: logicalY --`
Return the y-direction pixel value given a logical y value as a float on the FP stack.

`: px>flx \ x -- ; F: -- flx`
Change from client pixels *x* to logical x units *flx*. The

`: bgc-Paint \ -- ;`
The Windows WM_PAINT message handler. Axes are draw first followed by the histogram and finally any "line and point" operations that have been recorded.

`: bgc-Create \ -- ;`
The Windows WM_CREATE message handler which sets up various default values.

`: bgc-SetMaxX \ x --`
The action performed in response bgset-maxX

`: bgc-SetMaxY \ y --`
The action performed in response bgset-maxY

19.3.5 Exported Words

The following words are exported from the `BaseGraphs` vocabulary into the `Forth` vocabulary. These are the only words required by an application in normal circumstances.

```
export BGraphControl    \ -- addr
export MaxBars          \ -- n
export BGS-ASKSELF      \ -- n
export BGM-SpellValue   \ -- n
export /BGSpell         \ -- n
export bgs.Value        \ addr1 -- addr2
export bgs.Abscissa     \ addr1 -- addr2
export bgs.Text         \ addr1 -- addr2
export bgSet-MaxX       \ n hCtl --
export bgSet-MaxY       \ n hCtl --
export bgGet-MaxX       \ n hCtl --
export bgGet-MaxY       \ n hCtl --
export bgSet-XDP        \ n hCtl --
export bgSet-YDP        \ n hCtl --
export bgGet-XDP        \ n hCtl --
export bgGet-YDP        \ n hCtl --
export bgSet-BarWidth   \ n hCtl --
export bgGet-BarWidth   \ n hCtl --
export bgSet-BarCount   \ n hCtl --
export bgGet-BarCount   \ hCtl -- n
export bgAdd-Value      \ n hCtl --
export bgSet-BarValue   \ val idx hCtl
export bgGet-BarValue   \ idx hCtl -- val
export bgAdd-Bar        \ n hCtl -- index
export bgSet-BarLbl     \ cadr ulen index hCtl --
export bgSet-BarColour  \ COLOUR index hCtl --
export bgGet-BarColour  \ hCtl index -- COLOUR
export bgMaxFreq        \ hCtl -- n
export bgModeVal        \ hCtl -- n
export bgShowCum?       \ hCtl -- flag
export bgGet-POI        \ hCtl -- POI
export bgGet-TooMany    \ hCtl -- flag

export bgClearHisto     \ hCtl --
export bgClearPoints    \ hCtl --
export bgNormHisto      \ hCtl --
export bgNormHisto(3)   \ hCtl --
export bgFitHeight      \ hCtl --
export bgFitWidth       \ hCtl --

export bgAt             \ x y hBG --
export bgLine           \ x y hBG --
export bgPoint          \ x y hBG --
export bgCircle         \ x y hBG --
export bgSquare         \ x y hBG --
export bgCross          \ x y hBG --
export bgCross2         \ x y hBG --
export bgDiamond        \ x y hBG --
export bgPenColour      \ colour hBG --
export bgLineWidth      \ width hBG --
export bgPointsize      \ radius hBG --
export bgSet-XLabel     \ caddr ulen hCtl --
```

```
export bgGet-XLabel     \ hCtl -- caddr ulen
export bgSet-YLabel     \ caddr ulen hCtl --
export bgGet-YLabel     \ hCtl -- caddr ulen
export bgGet-BarLbl     \ hCtl n -- caddr ulen

\ export bgHistogram   \ hCtl -- ; set to (non-numbered) histogram
\ export bgDotted      \ hCtl -- ; set to (non-numbered) dotted histogram
\ export bgNumbered    \ hCtl -- ; set to a histogram to numbered
\ export bgCartesian   \ hCtl
export bgmHistogram     \ -- n
export bgmNumbered      \ -- n
export bgmDotted        \ -- n
export bgmCartesian     \ -- n
export bgmWithPlots     \ -- n
export bgSet-Mode       \ n hCtl --
export bgGet-Mode       \ hCtl -- n
export bgShowCum        \ hCtl --
export bgHideCum        \ hCtl --
export bgSet-POI        \ n hCtl -- ; 0 <=n <=1000
export bgSet-TooMany    \ hCtl
export bgPlotPercents   \ flag hCtl --
export bgPlotPercents?  \ hCtl -- flag
export bgShowGrid       \ flag hCtl --
export bgGridShown?     \ hCtl -- flag
export bgSetGridColour  \ colour hCtl --
export bgGridColour     \ hCtl -- colour
export bgShowMean       \ flag hCtl -- )
export bgMeanShown?     \ hCtl -- flag
export bgSetMeanColour  \ colour hCtl --
export bgMeanColour     \ hCtl -- colour
export bgSetMean        \ val hCtl --
export bgMean           \ hCtl -- val

[defined] floats [if]
export bgSetNormDist    \ hCtl --   F: f1 f2 -- ; F: var mean --
export bgGetNormDist    \ hCtl --   F: -- f1 f2 ; F: -- var mean
export bgShowNormDist   \ flag hCtl --  F: --
export bgNormDistShown? \ hCtl -- flag ;    F: --
export bgSetNDistClr    \ colour hCtl --
export bgNDistClr       \ hCtl -- colour
[else]
cr
.( ******************************************************) cr
.(              Base Graph is defined without floating point) cr
.(              and so the Base Graph Normal Distribution   ) cr
.(              Display functions have not been compiled.   ) cr
.( ******************************************************) cr
[then]
```

The following shows two simple examples one of displaying a histogram and one of showing a line graph. Type `TestHisto` and `TestPlots` to see these examples.

```
NextID: idc_BGraph

dialog: GraphDlg
    50 10 300 100 Position&Size
    Caption "Testing the Base Graph control"
    WS_VISIBLE WS_CAPTION or Style
    WS_SYSMENU Style+
    DS_CENTER Style+
    WS_ThickFrame Style+
    DS_3DLOOK Style+
    DS_SetFont Style+

    10 100 0 Font "Arial"

    5 5 150 120     idc_BGraph  zNull   BGraphControl

    WM_Size does:   0 0 CurrWin ClientSize
                    idc_BGraph CtlHdl Set-WinDims
            ;

end-dialog


0 value hDlg
0 value hBG

: GD
    0 GraphDlg ShowModeless Set-Currwin
    CurrWin to hDlg
    idc_BGraph ctlhdl to hBG
;

: MkLbls    \ --
    hBG bgGet-BarCount 0 ?do
    i [char] A + pad c!
    pad 1 i hBG bgSet-BarLbl
    loop
;


: TestHisto \ --
    gd hbg set-currwin
    s" Testing Histogram" hDlg Set-WinText
    100 hBG bgSet-MaxX
    100 hBG bgSet-MaxY

    15 hbg bgSet-BarWidth
    10 hbg bgSet-BarCount
    hBG bgFitWidth
    MkLbls
    300 0 do 100 choose hbg bgAdd-Value loop
```

```
;

: PlotLine  \ x y hCtl --
    3dup bgLine bgPoint
\ bgLine
\ bgSquare
\ 3dup bgCircle 3dup bgSquare bgCross
;

: TestPlots
    only forth also BaseGraphs
    gd hbg set-currwin

    bgmCartesian hbg bgSet-Mode

    Blue hBG bgPenColour
\ 10 hBG bgPointsize   \ radius hBG --

    5 10 hBG bgAt
    5 10 hBG bgPoint
    10 25 hBG PlotLine
    20 50 hBG PlotLine
    30 70 hBG PlotLine
    40 85 hBG PlotLine

    red hBG bgPenColour
    2 hBG bgLineWidth
    50 80 hBG PlotLine
    60 65 hBG PlotLine
    70 55 hBG PlotLine
    80 40 hBG PlotLine
    90 22 hBG PlotLine
    s" The x axis is 'ere" hBG bgSet-XLabel \ caddr ulen hCtl --

    hBG bgFitWidth
;
```

20 The GUIgen PieControl

20.0.1 Introduction

This code provides basic pie charting through `PieControl` a GUIgen control. *Pie*s are depicted as one-dimensional circles split into coloured *slices* as required. Options include placing a title and/or a caption on the chart and the labelling of individual slices.

Here is an example of using a `PieControl` where the 'pie' is made to show 5 'slices' in the default colours.

```
NextID: idc_Pie

Dialog: PieBox
    150 150 Size
    caption "Testing the Pic Control"
    WS_VISIBLE WS_CAPTION or Style
    DS_CENTER Style+
    WS_ThickFrame Style+
    DS_3DLOOK Style+
     10 100 0 Font "Arial"

    5 5 120 120     idc_PIE  zNull   PieControl

    WM_InitDialog does:
        idc_Pie CtlHdl >r

        5 r@ pieCount!
        60 0 r@ pieSlice!
        40 1 r@ pieSlice!
        30 2 r@ pieSlice!
        11 3 r@ pieSlice!
        8  4 r@ pieSlice!
        r> drop
    ;
end-dialog

0 PieBox ShowModeless
```

20.0.2 Setting up

The Pie Control code is compiled into its own directory `PieControls`. Application words are exported using the VFX `Export` facility to the vocabulary into which other GUIgen public words are exported to.

`vocabulary PieControls`
The vocabulary into which the the Pie Control code is compiled.

20.0.3 The PieClass Window class and its Data Structures

The `PieControl` is based in the Windows class `PieClass` defined below. `PieClass` 'contains' the *WinProps* data structures used to create and maintain the Pie Control.

An extended WinProps structure `PClassProps` is defined as the WinProps for each `PieClass`. I contains various property values including an array of *slices* - each slice containing data defining how it depicted within the pie's bounding circle.

`struct /slice`

The structure defining how a pie slice is drawn. It consists of the followig fields:

```
    cell field  slice.Val           \ frequency/value
    cell field  slice.CumProp       \ proportion or total * 100
    cell field  slice.CumAngle      \ cumulative angle.
    24   field  slice.Label         \ 24 character text label
    cell field  slice.displaced     \ is this slice displaced slightly
    cell field  slice.colour        \ used when table colours not used
```

`20 constant MaxSlices`

The maximum number of slices which can be contained in a `PieControl`.

`winprops: /PClassProps`

The `PieClass` WinProps structure. Defiend as:

```
    0               field   bpc.PropStart   \
    cell            field   bpc.Count       \ number of slices shown
    cell            field   bpc.TooMany     \ true if tried to add too many
    cell            field   bpc.Total       \ Total: Sum of slices
    cell            field   bpc.CurrSlice   \ address of slice being drawn
    /Slice          field   bpc.Slice[-1]   \ dummy slice to allow 'backward'
                                            \ compare
    MaxSlices /slice array-of bpc.slices    \ the slices array
    cell            field   bpc.Labelled?   \ true if labels being used...
    2 cells         field   bpc.Centre      \ x and y centre position
    cell            field   bpc.radius      \ radius of pie
    RECT            field   bpc.Bounds      \ bounding rectangle of pie
    64              field   bpc.title       \ title of pie - shown at top
    64              field   bpc.Caption     \ caption of pie - shown at bottom
end-struct
```

The `PieClass` is defined as follows:

```
WinClass: PieClass
    /PClassProps PropertySize

    Name "GUIgen-BasePIEClass"
    CS_HREDRAW CS_VREDRAW or    Style
    CS_DBLCLKS                  Style+
end-winclass
```

PieControl Styles

A Single PieClass/PieControl styles is defined:

`1 constant PCS-SliceDisplace`
PieControls with this style show individual slices slightly displaced if the mouse pointer is positioned over a slice and the mouse button is clicked.

PieControl Slice Colours

Each PieControl slice is a different colour to its neighbour. A table of ten colours are defined which are used by default but an application can define its own set of ten colours.

`Create <PieColours>`
An set of ten RGB colour values used by default when colouring a histogram. Defined simply as :

```
    $FF8080 ,    \ PURPLEY
    $FF7FFF ,    \ LT PURPLE
    $C0FFFF ,    \ CREAM
    $80FFA0 ,    \ PASTEL GREEN
    $40C000 ,    \ GREEN
    $8080FF ,    \ DUSKY PINK
    $FF8000 ,    \ PASTEL BLUE
    $00C0C0 ,    \ YELLOWY
    $0080FF ,    \ ORANGE
    $0000FF ,    \ RED
```

`10 cells buffer: <UserColours>`
An aray of ten cells the value of which are intended to be set by an application.

`defer PieColours    \ -- addr`
The address returned is either that of <PieColours> or of <UserColours>. This is the word used to select which colours are used to colour in pie slices.

`: DefaultColours    \ --`
Sets PieColours to return the address of <PieColours>. The default behaviour.

`: UserColours       \ --`
Sets PieColours to return the address of <UserColours>

`: SliceColour[]     \ n -- addr`
Returns the address of n-th colour value from the set of ten cells given by PieColours.
Use the following steps to define a set of user colours. (But note that the method is subject to change. It's rubbish!)

- Add PieControls to the search order.
- Execute UserColours
- Set the zeroth colour value for instance `red 0 SliceColour[] !`
- Set the other nine colours in a similar way.
- Remove PieControls from the search order.

20.0.4 Adding and Setting data to a PieControl.

Setting Data

The following glossary lists the application words used to set a `PieControl`'s properties. The items in stack comments have the following meanings:

n A standard cell-sized integer.

hPie The Windows handle of a `PieControl`.

slice# An integer indicating the index number of a slice.

caddr ulen A text string; an address/count pair.

bool A cell-sized boolean flag - TRUE or FALSE

`: pieCount!  \ n hPie -- ;`
Set number of slices of a `PieControl` to *n*.

`: pieSliceInc  \ slice# hPie -- ;`
Add 1 to a slice's value

`: pieSlice+  \ n slice# hPie -- ; add n to a slice`
Add n to a slice's value

`: pieSlice!  \ n slice# hPie -- ; set slice value to n`
Set a slice's value to n.

`: pieSetTitle  \ caddr ulen hPie --`
Set the title of a Pie Control. The Title is shown at the top of the control.

`: pieSetCaption \ caddr ulen hPie --`
Set a Pie Control's caption. The caption is shown at the bottom of the control.

`: pieSetLabel  \ caddr ulen slice# hPie --`
Set the label text for a particular slice.

`: pieSliceDisplace  \ bool slice# hPie --`
Causes a slice to be shown 'displaced' if *\(bool} is True.

`: pieClear  \ hPie --`
Clear the Pie Control. That is the control's *\fo/PClassProps} fields are all cleared/erased.

`: pieColour!  \ n slice# hPie -- ; set slice colour to n`
Set the colour of the specified slice to n. n is a RGB colour value.

`: pieTooMany  \ hCtl --`
sets the 'Too many' property to True.

`: pieExplode  \ flag hPie --`
Display all slices of the pie as (a) displaced if *flag* is TRUE or (b) undisplaced if *flag* is FALSE. Performs `pieSliceDisplace` for each slice.

`: pieSliceDispOnly  \ slice# hPie -- ;`
Displace only the given slice.

Getting Data

The following glossary lists the application words used to obtain data from a `PieControl`.

`: pieCount@  \ hPie -- n ;`
Return the number of slices of the Pie Control with Windows handle *hPie*.

```
: pieSlice@     \ slice# hPie -- n ;
```
Return the value of slice number *\{slice#}.

```
: pieXY>Index   \ X Y hPie-- slice#
```
Given the *X* and *Y* coordinates in the given Pie Control, return the number (index) of the slice over which the point (X,Y) lies.

```
: pieSliceLabel     \ idx hPie -- caddr ulen
```
Return the text label associated with slice number *idx*.

```
: pieSliceColour    \ idx hPie -- COLOUR
```
Return the RGB colour of slice number *idx*.

```
: pieSlice%     \ idx hPie -- percent*100
```
Return the value as a percentage, of slice number *idx*. Note that the returned value is scaled by 100, so a value returned of 123 indicates 1.23%.

```
: pieAddSlice   \ val ca u hPie -- ;
```
Add slice of val value and label ca/u

```
: .pie      \ hPie --
```
For debugging/testinmg purposes, this word lists a Pie Control's slice values to the console.

Index

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z